MASTER EN SCIENCES INFORMATIQUES

Elaboration d'une méthode outillée permettant l'analyse du cycle de vie des requêtes de systèmes logiciels dits "data-intensive"

Ballarini, David; Vondou, Guessing

Award date:
2016

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2015–2016

Élaboration d'une méthode outillée permettant l'analyse du cycle de vie des requêtes de systèmes logiciels dits "data-intensive"

David BALLARINI | Guessing VONDOU

Maître de stage : Romain ROBBES

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Anthony CLEVE

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

De nos jours, l'omniprésence et l'impact des systèmes logiciels dits data-intensive leur confèrent une place prépondérante dans le fonctionnement de notre société. Pour préserver leur valeur marchande (e.g., la valeur économique, la satisfaction des parties prenantes) et ne pas devenir obsolètes dans un environnement en perpétuelle évolution, ces systèmes doivent répondre à un ensemble de changements tout au long de leur cycle de vie. Comprendre les changements passés permet alors de mieux construire le présent et le futur du système. Une des manières de comprendre ce passé consiste à récupérer et analyser les riches ensembles de données contenues dans les différents dépôts logiciel, afin de découvrir de nouvelles informations utiles. L'extraction automatique des requêtes d'accès à la base de données de versions historiques d'un système, combinée à des techniques d'analyse d'origine d'artefacts du code source, permettent alors de proposer une méthode analysant le cycle de vie de ces requêtes. Cette contribution est construite grâce à la combinaison entre des travaux antérieurs liés à l'analyse statique et l'extraction automatique de requêtes, et un apport original sur l'analyse d'origine de requêtes. La méthode proposée est ensuite outillée et évaluée sur des systèmes open-source en développement depuis plusieurs années. Finalement, les limites de l'évaluation sont capturées ainsi que les perspectives futures d'utilisation d'une telle méthode.

Nowadays, software systems got a deep impact on their environment, allowing them to play a key role in our society. In order to preserve their value on the market (e.g., economic value, stakeholders' satisfaction), we have to be willing to face changes for our software systems running in a world that keeps moving forward. Understanding changes from the past allows to build a better system for the future. A way to understand the past consists in gathering and analyzing rich data that can be found in various software repositories, in order to get a useful set of new data. The automatic mining of database requests from historical versions of a software system, combined with techniques used in the field of origin analysis, allows us to establish a new method which analyzes the lifecycle of database requests. This contribution is build thanks to the combination between previous work related to automatic mining of database requests, and an original contribution about origin analysis of database requests. The method is then implemented and evaluated on open-source software whose development stage has been carried out for many years. Finally, limitations are determined as well as potential future uses.

Avant-propos

Le travail présenté dans ce mémoire représente l'aboutissement de recherches effectuées lors d'un stage effectué à l'Université du Chili (plus particulièrement au sein du Département des Sciences Informatiques de la Faculté des Sciences Physiques et Mathématiques), sur une durée de 3 mois. Il s'agit de recherches établies dans le cadre du programme de Master en Sciences Informatiques proposé à l'Université de Namur. Ces recherches portent sur l'analyse de données historiques issues de plusieurs systèmes open-source, ces derniers ayant été développés et utilisés depuis plusieurs années en situation réelle à divers endroits du globe. Ce stage a été effectué sous la supervision locale du Professeur Romain Robbes, ainsi que sous la supervision extérieure de notre promoteur Anthony Cleve.

Nous aimerions remercier tout d'abord et de manière particulière, Romain Robbes et Anthony Cleve dont le dévouement, l'encadrement continu et les précieux conseils nous ont permis de faire dérouler le stage et la rédaction de ce mémoire dans des conditions de travail optimales. Leur motivation et leur passion pour la recherche dans leurs domaines respectifs nous ont été transmises avec enthousiasme et optimisme. Ensuite, nous aimerions également remercier Loup Meurice pour ses nombreux soutiens lors de nos recherches et de nos éventuelles questions relatives à ses travaux. Finalement, nos remerciements s'adressent également à toutes les autres personnes ayant gravité de près ou de loin au déroulement de nos activités de recherche, telles que le personnel du Département des Sciences Informatiques à Santiago, ou encore Csaba Nagy pour les réponses apportées à nos questions techniques sur ses travaux.

Pour conclure, nous remercions également nos familles pour leur soutien apporté durant l'entièreté de notre parcours académique, ainsi que pour leurs précieuses relectures de ce travail.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Évolution des systèmes logiciels

De nos jours, l'omniprésence et l'impact des systèmes logiciels leur confèrent une place prépondérante voire même un rôle-clé dans le fonctionnement de notre société. Pour préserver leur valeur marchande (e.g., la valeur économique, la satisfaction des parties prenantes) et ne pas devenir obsolètes dans un environnement en perpétuelle évolution, ces systèmes doivent répondre à un ensemble de changements tout au long de leur cycle de vie. Ceux-ci peuvent par exemple être dus à une modification des exigences requises, à l'évolution de l'environnement du système ou à l'identification d'erreurs. Cette ensemble de changements fait partie du processus d'évolution et de maintenance du système logiciel.

Les activités de maintenance et d'évolution du logiciel sont d'une importance capitale pour la communauté du génie logiciel. En effet, plusieurs études indiquent qu'au moins 50% du budget de développement total d'un logiciel est consacré à la maintenance logicielle, ce nombre pouvant même dépasser les 90% dans certains cas [1, 2, 3]. En outre, il est estimé que d'ici 2020 seulement 30% du travail des développeurs sera consacré à des nouveaux projets, alors que les 70% restants seront destinés à la maintenance et l'évolution de systèmes existants [4].

Pourtant, ces activités étaient (jusqu'à récemment) souvent sous-évaluées dans les processus de développement logiciel. Dans les modèles traditionnels (e.g., le *modèle en cascade*), tous les efforts des développeurs sont typique-

ment concentrés sur le développement initial du logiciel, qui s'occupe du développement de la première version fonctionnelle du logiciel. La phase de maintenance est quant à elle une activité réalisée exclusivement après le déploiement du produit logiciel, et se cantonne essentiellement à des modifications mineures et des corrections de bogues. Cette vision très rigide implique que les exigences du client soient fixées dès le début du projet, alors que celles-ci évoluent communément pendant tout le cycle de vie logiciel.

De manière à jouir d'un cycle de développement logiciel efficace et durable, il est dès lors nécessaire de placer l'évolution logicielle au centre du processus de développement logiciel [5]. De plus, un des objectifs de la communauté du génie logiciel vise à fournir davantage et de meilleurs supports pour assister les développeurs dans leurs tâches d'évolution du système [5]. Cette nécessité est d'autant plus pertinente pour faire face à la complexité accrue des logiciels dits "*data-intensive*" présentés dans la section suivante.

1.2 Évolution des logiciels "data-intensive"

Les logiciels dits "*data-intensive*" sont des logiciels où la conservation, le traitement et l'analyse de très grands ensembles de données sont la préoccupation majeure à gérer. A contrario des applications dites "*compute-intensive*" dont l'objectif est de centrer majoritairement leur temps d'exécution sur du calcul interne, les applications *data-intensive* se concentrent principalement sur le traitement des entrées/sorties des données et la manipulation de ces dernières. Ces logiciels interagissent dès lors fortement avec une base de données, dans laquelle sont stockées les données liées au domaine d'application du système. Cette dernière est typiquement gérée par un système de gestion de base de données, structurée selon un schéma qui modélise fidèlement l'organisation des données et les relations entre elles.

Selon Mattman et al. [9], ces logiciels se sont multipliés depuis le début du 21$^{\text{ème}}$ siècle étant donné la motivation pour beaucoup d'acteurs issus de domaines différents de pouvoir gérer des ensembles de données toujours de plus en plus importants. Cette tendance est donc devenue globale et largement soutenue, notamment dans des domaines où le traitement de ces données peut apporter une plus-value non négligeable. Parmi ceux-ci, le domaine de la recherche médicale est par exemple concerné, ces logiciels apportant dès

lors un outil de contribution idéal.

Ce volume croissant de données, lors de l'évolution du système, implique donc de mettre en place des mécanismes adéquats pour gérer cette volumétrie durant l'entièreté du cycle de vie du logiciel. Gorton identifie donc un besoin imminent afin de mettre en place des technologies permettant la gestion des métadonnées et leur analyse, ces technologies devant être adaptées à la croissance de complexité du système au cours du temps [10].

Pour gérer l'évolution de tels systèmes, il n'est pas suffisant de comprendre le programme d'application. En effet, il est également nécessaire d'avoir une maîtrise du système de données associé, et des interactions avec celui-ci. Une technique efficace pour comprendre un système et ses interactions est d'analyser son historique de développement. La section suivante expose les bénéfices de cette approche.

1.3 Analyse de l'historique du système

> *Ceux qui ne peuvent se souvenir du passé sont condamnés à le répéter.*
>
> — George Santayana

Analyser l'historique d'un système permet souvent de recueillir des informations additionnelles sur sa version courante. Des changements mal maîtrisés peuvent avoir des répercussions considérables sur l'entièreté du système, et peuvent alors engendrer des coûts élevés pour maîtriser leurs impacts. Afin de réduire ce risque, il est essentiel de déterminer et de comprendre les tenants et aboutissants des changements particuliers ayant donné lieu à la forme actuelle du système. Comprendre et analyser le passé est primordial afin de construire un futur sain. De la même façon, l'analyse de l'historique de l'évolution du système vise à fournir aux développeurs un support supplémentaire permettant une prise de décision plus informée, avec pour objectif de faire évoluer le système d'une manière cohérente, robuste et pérenne.

Il s'agit d'un des principes sur lequel repose le champ du *Mining Software Repositories*, où sont analysées les larges quantités de données présentes dans les différents dépôts liés au logiciel. De nombreuses recherches ont étudié

l'historique d'évolution de systèmes logiciels à partir d'artefacts variés comme le code source, les mailing lists de développement, la documentation, les rapports de bogues, l'enregistrement de conversations, etc. Dans ce travail, les techniques de *Mining Software Repositories* vont être utilisées pour découvrir des nouvelles informations utiles sur la base de données de systèmes *data-intensive*.

1.4 Contribution

Dans cette thèse nous proposons une approche permettant de détecter, d'analyser et de classifier l'évolution des accès vers la base de données au long du cycle de vie de systèmes logiciels *data-intensive*. Cette approche semi-automatique vise à assister les développeurs dans des tâches de mining de dépôt logiciel. L'approche est basée sur le concept d'analyse d'origine afin de reconstruire le cycle de vie des requêtes extraites depuis l'historique du système. Nous proposons une implémentation de notre méthodologie dans une méthode outillée complète et fonctionnelle, validée par une application à de larges systèmes *data-intensive*.

1.5 Structure de la thèse

Le chapitre suivant passe en revue l'état de l'art relatif à l'évolution logicielle, à l'analyse de l'historique de développement et aux techniques d'analyse pour supporter l'évolution de systèmes *data-intensive*. Dans le Chapitre 3, nous présentons les concepts préliminaires nécessaires à la compréhension de notre approche. Le Chapitre 4 détaille notre approche pour analyser l'évolution des requêtes d'accès présentes dans l'historique de développement de systèmes *data-intensive*. Nous présentons les heuristiques utilisées dans notre méthode outillée, les choix d'implémentation réalisés et les fonctionnalités développées. Le Chapitre 5 présente une étude de cas démontrant l'utilisation de la méthode outillée. Une évaluation y est réalisée sur base de la méthode outillée, et les résultats obtenus sont discutés, ainsi que les limites de l'évaluation. Enfin, le Chapitre 6 conclut notre travail par une rétrospective des points abordés tout au long de ce mémoire, suivie d'une description des contraintes et limitations observées et finalement d'une réflexion sur les perspectives de recherches futures.

Chapitre 2

État de l'art

Un très grand nombre d'études ont été réalisées dans le domaine de l'évolution logicielle. Les sections suivantes présentent plusieurs études et concepts relatifs à notre approche, matérialisée par un outil d'analyse du cycle de vie des requêtes d'accès à une base de données.

2.1 Évolution des systèmes logiciels

Manny Lehman est l'un des pionniers de l'évolution logicielle. Il a étudié l'évolution à long terme de plusieurs systèmes logiciels d'IBM entre 1969 et 2001, ce qui lui a permis de formuler huit lois relatives à l'évolution logicielle [6, 7, 8]. Ces huit règles s'appliquent aux logiciels qui sont destinés à résoudre des problèmes du monde réel, caractérisés par des exigences fortement affectées par leur environnement. Ces lois sont les suivantes :

1. Modification continue : Un logiciel doit être continuellement adapté ou il deviendra progressivement moins satisfaisant.

2. Complexité croissante : Au fur et à mesure qu'un logiciel est modifié, il devient de plus en plus complexe, à moins qu'un travail soit effectué pour réduire cette complexité.

3. Autorégulation : L'évolution logicielle est un processus autorégulateur, dont la distribution des artefacts liés au processus ou au produit est

proche d'une distribution *normale*. Toute mesure réalisée sur le produit ou sur le processus logiciel (e.g., la taille du logiciel, le nombre de bogues signalés etc.) ne varie donc pratiquement pas entre deux versions successives.

4. Conservation de la stabilité organisationnelle : Pendant la durée de vie d'un programme, sa fluctuation d'activité est très peu variable : la charge de travail associée à chaque version est presque constante.

5. Conservation de familiarité : La quantité de contenu ajoutée dans chaque version successive d'un système logiciel a tendance à rester constante ou à diminuer. Cela signifie que de manière à conserver la maîtrise du contenu et du comportement du système logiciel, ses développeurs devraient le faire évoluer par incréments fixes.

6. Croissance continue : Le nombre de fonctionnalités d'un logiciel doit continuellement augmenter pour maintenir la satisfaction des utilisateurs tout au long de son cycle de vie.

7. Diminution de qualité : Sans contre-mesures actives, la qualité d'un logiciel se dégrade graduellement quand le système évolue.

8. Système de feedback : Le processus de développement d'un logiciel est un système dont le comportement est contraint par des boucles de feedback complexes. Ces boucles de feedback complexes sont créées suite au développement continu du logiciel. Le feedback peut venir de plusieurs sources, ce qui inclut l'environnement technique, le domaine d'application, les utilisateurs ou le système lui-même en cas d'identification de défauts qui nécessitent une correction.

Les lois de Lehman ont été formulées sur base de plusieurs études empiriques de larges systèmes développés dans un environnement industriel, avec des équipes bien définies, et suivant typiquement un processus de développement rigide de type *modèle en cascade*. Cependant le monde logiciel a fortement évolué depuis ces études, notamment avec l'établissement des pratiques de développement *open-source*. Plusieurs auteurs ont depuis lors

montré qu'une partie de ces lois ne s'appliquent pas dans plusieurs cas, particulièrement dans les systèmes *open-source* [11, 12]. Ces lois restent pourtant très influentes dans le domaine de l'évolution logicielle : elles permettent de comprendre les effets et les risques qui entourent les changements appliqués à un logiciel au cours du temps, et les mesures à suivre pour maîtriser son évolution.

2.2 Analyse des dépôts logiciels

L'historique de développement d'un système logiciel contient des informations très riches sur les changements effectués pendant son évolution. Grâce aux techniques et outils de *Mining Software Repositories*, il est possible d'exploiter ce grand ensemble de données pour réaliser une pléthore de tâches comme analyser les clones de code, faire de la prédiction de bogues, analyser la coévolution entre différents artefacts etc. Il s'agit donc d'un support de choix pour mieux comprendre un système et améliorer sa qualité.

Les objectifs principaux du *Mining Software Repositories* sont [13] :

— Découvrir ou confirmer des faits à propos de systèmes logiciels et de leur évolution.

— Assister les développeurs, managers, testeurs etc. qui travaillent sur ces systèmes.

— Réaliser des prédictions sur le futur de systèmes logiciels.

Il existe de nombreuses approches pour réaliser ces objectifs, qui diffèrent principalement par les métriques utilisées dans leur modèle respectif. Une métrique logicielle est définie comme étant une mesure du degré avec lequel un composant ou processus logiciel possède un certain attribut. Les métriques peuvent typiquement être classifiées en deux groupes, selon la source de données utilisée.

D'une part il existe les métriques dites centrées sur le produit, qui sont calculées à partir des données présentes dans un "instantané" du logiciel.

Ces mesures portent typiquement sur la structure statique ou dynamique du code source comme le nombre de lignes de code, la complexité cyclomatique, le couplage entre deux entités, le *fan-in* (i.e., le nombre de modules qui appellent un module donné), le *fan-out* (i.e., le nombre de modules appelés par un module) etc.

D'autre part il y a les métriques dites centrées sur le processus, qui sont quant à elles calculées à partir de données extraites depuis les artefacts liés au processus (e.g., à partir du gestionnaire de version). Par exemple le nombre de révisions, le nombre de lignes ajoutées ou supprimées entre deux changements, ou le nombre d'auteurs qui ont soumis un fichier dans un dépôt. Il s'agit bien ici du type de métrique qui va tirer parti des informations accumulées dans l'historique de développement du système logiciel.

Plusieurs artefacts sont produits pendant le cycle de développement d'un logiciel. Les plus utilisés dans la littérature pour étudier l'historique d'évolution logicielle sont généralement le code source, les dépôts de gestion de versions, les dépôts de suivi de bogues et les archives de communication. Un dépôt logiciel désigne l'endroit où sont stockées les modifications réalisées sur le code source à l'égard d'un thème commun, ainsi que des méta-informations liées à ces modifications. La quantité d'informations stockées dans les dépôts peut rendre leur analyse extrêmement complexe. En effet, il n'est pas rare de trouver des systèmes commerciaux vieux de plusieurs décennies, qui ont accumulé un grand nombre de versions pendant leur cycle de vie. Il est donc essentiel de sélectionner uniquement les données pertinentes compte tenu de la tâche considérée.

De nombreuses recherches ont été effectuées pour étudier l'évolution logicielle au moyen de l'historique de développement. Une grande variété de sujets ont été abordés, tels que la prédiction de défauts, la détection de dépendances entre des fragments de code, la duplication de code ou encore la gestion de l'évolution conjointe entre plusieurs artefacts dépendants. Les points qui suivent introduisent plusieurs études représentatives du domaine.

2.2.1 Prédiction de défauts

Moser et al. [16] ont comparé le pouvoir prédictif d'un ensemble de métriques liées d'une part au produit et d'autre part au processus pour prédire les défauts présents dans plusieurs versions du logiciel Eclipse. Les activités d'assurance de qualité logicielle (e.g., les tests unitaires, l'inspection de code etc.) constituent un pan important du génie logiciel. Les ressources mises à disposition par l'entreprise (e.g., le temps, le personnel etc.) étant limitées, il semble essentiel d'optimiser l'efficacité de celles-ci. La prédiction de défauts a pour but d'identifier les modules particulièrement disposés à contenir des défauts. Cela permet de donner la priorité à la révision des modules les plus exposés et donc d'améliorer la gestion des ressources disponibles.

Leur approche analyse l'utilisation des métriques dans le cas d'une tâche de classification binaire au niveau des fichiers, c'est-à-dire dans le but de prévoir la présence ou l'absence de défauts dans un fichier. Dans une étude précédente, Zimmermann et al. [17] montraient que les métriques de code produisent des performances prédictives prometteuses quant à la classification des packages comme défectueux ou non, mais des résultats plus nuancés lorsque la granularité s'affine au niveau des fichiers. Moser et al. ont créé un modèle prédictif basé sur des métriques de changement (i.e., calculées à partir de l'historique de changement des fichiers) et un autre basé sur une combinaison des deux types de métrique. Ils ont ensuite comparé leurs modèles avec celui utilisé par Zimmermann et al. .

Un grand ensemble de métriques de changement a été considéré. Cet ensemble inclut notamment le nombre de révisions d'un fichier, la taille et l'âge du fichier, et le nombre de fois qu'un fichier a été impliqué dans des activités de correction de bogues. Cette dernière métrique est calculée à partir des commentaires du gestionnaire de version au moyen d'un algorithme trivial de *pattern matching*. Cela permet de simplifier l'approche utilisée, en considérant comme unique artefact l'historique de changement (et non un dépôt de suivi de bogues), au détriment de sa fiabilité.

Chaque modèle de classification a été construit en utilisant trois approches d'apprentissage automatique : la régression logistique, la classification naïve bayésienne et les arbres de décision. Leurs résultats indiquent que les mo-

dèles basés sur les métriques de changement surpassent nettement les prédicteurs basés sur des attributs statiques du code source pour les données d'Eclipse analysées, quelle que soit l'approche d'apprentissage utilisée. Pour les auteurs, les métriques de complexité correspondraient davantage à un indicateur de l'effort cognitif mobilisé pour comprendre le code source, plutôt qu'à un indicateur des défauts potentiels. A contrario, les métriques de changement et en général les métriques liées au processus contiendraient des informations plus discriminatoires et significatives sur la distribution des défauts dans un système logiciel.

Ils ont également appliqué une version dite "sensible au coût" de leur approche. A la différence de l'approche standard, celle-ci prend en compte le coût associé aux différentes erreurs de prédiction des modèles. En effet, ne pas inspecter un fichier classifié à tort comme non-défectueux a des répercussions potentiellement plus graves que l'allocation de ressources à la vérification d'un fichier non-défectueux. Les résultats obtenus renforcent ceux observés dans la première approche.

Dans le même domaine, Eick et al. [18] ont étudié les facteurs qui expliquent l'érosion de code en analysant plus de 15 ans d'historique de changement d'un large système de télécommunications. Une portion de code est victime du phénomène d'érosion (ou “code decay”) si elle est plus difficile à changer que ce qu'elle devrait l'être. Cette difficulté peut se manifester par une augmentation du coût ou du temps nécessaire à l'implémentation du changement, ou par une diminution de la qualité du logiciel modifié.

Les auteurs ont identifié plusieurs causes à ce phénomène, qui sont toutes fondamentalement liées aux changements appliqués au code. Parmi celles-ci figurent une architecture inappropriée, la violation de principes originaux de design, des exigences imprécises, la pression du temps sur les développeurs, des outils de programmation inadéquats, un environnement organisationnel peu favorable, la variabilité des développeurs et un processus de changement inadéquat.

Ils ont également identifié plusieurs symptômes qui indiquent la manifestation de l'érosion de code. Ces symptômes incluent la complexité excessive du code, un historique de changements fréquents, un historique de défauts,

des changements largement dispersés, des *kludges* (i.e., du code bidouillé) et un nombre élevé d'interfaces.

Eick et al. proposent finalement une série d'indices d'érosion du code, calculés directement à partir du gestionnaire de version. Il s'agit de l'historique des changements fréquents, la portée des changements, la taille du module, l'âge du module, le potentiel d'erreur d'un module et l'effort nécessaire à l'implémentation d'un changement. Ces indices permettent aux développeurs de prédire l'apparition d'une érosion de code, et donc de prendre des mesures pour éviter celle-ci.

2.2.2 Détection de dépendances implicites

Zimmermann et al. [19] ont appliqué des techniques de *data mining* à l'historique des versions de plusieurs systèmes afin de guider les développeurs lorsqu'ils effectuent des changements susceptibles de cacher des dépendances implicites. Les raisons de ces dépendances implicites, désignées sous le nom de *couplage évolutionnaire*, peuvent être multiples : elles peuvent être dues à la duplication de fragments de code, ou être la conséquence d'autres problèmes liés au design. Leur approche repose sur l'intuition que des entités qui ont été modifiées ensemble dans le passé ont de grande chances d'être modifiées ensemble dans le futur. Explorer les archives de version d'un système logiciel permettrait donc d'obtenir davantage d'informations sur les dépendances entre les entités, comparativement à une analyse du programme.

Les auteurs ont développé un outil baptisé *ROSE* qui répond aux objectifs suivants :

— Suggérer et prédire des futurs changements.

— Détecter entre des entités un couplage non détectable par une analyse du programme.

— Prévenir les erreurs induites par des changements incomplets.

La première étape de leur approche consiste à explorer les archives de

gestion de versions afin d'extraire les changements et les transactions relatifs aux entités considérées (e.g., les méthodes, les classes, les fichiers etc.). Un changement correspond à la modification, l'ajout ou la suppression d'une entité ; tandis qu'une transaction regroupe un ensemble de changements simultanément soumis par un développeur à un gestionnaire de version.
A partir des transactions, l'objectif de *ROSE* était d'inférer des règles qui associent pour chaque changement d'autres changements potentiellement liés. Chaque règle possède une probabilité de survenance déterminée par les deux métriques suivantes :

1. Le nombre d'appuis : Il s'agit du nombre de transactions à partir desquelles la règle a été inférée. Soit la fréquence d'un ensemble x dans un ensemble de transactions D calculée comme suit :

$$freq_D(x) = |\{t \mid t \in D, x \subseteq t\}| \tag{2.1}$$

   Le nombre d'appuis d'une règle $x_1 \Rightarrow x_2$ pour un ensemble de transactions D est défini comme :

$$count_D(x_1 \Rightarrow x_2) = freq_D(x_1 \cup x_2) \tag{2.2}$$

2. La confiance : Cette mesure détermine le poids de la conséquence d'une règle par rapport à l'ensemble des alternatives associées à un antécédent. Elle est calculée comme suit :

$$conf_D(x_1 \Rightarrow x_2) = \frac{count_D(x_1 \Rightarrow x_2)}{freq_D(x_1)} \tag{2.3}$$

Une version optimisée de l'*Apriori Algorithm* est ensuite utilisée pour calculer les règles d'association [20]. L'algorithme prend en entrée un nombre minimum d'appuis et une confiance minimum pour ensuite calculer l'ensemble des règles d'association qui sont supérieures aux deux seuils. Une fois qu'un développeur soumet des changements au gestionnaire de version, ceux-ci sont ajoutés en tant que transactions. Basé sur ces transactions et sur les règles

d'association, *ROSE* calcule la liste de suggestions qui sera proposée, et l'ordonne par ordre de confiance.

Zimmermann et al. ont testé leur approche avec les archives de huit larges systèmes *open-source*. Ils ont découvert que *ROSE* donne des suggestions nombreuses et précises pour les systèmes dans lesquels peu de nouvelles fonctionnalités sont introduites. Dans le cas de systèmes évoluant rapidement, la plupart des suggestions utiles sont au niveau des fichiers. Ce n'est pas surprenant puisque *ROSE* devrait alors prédire de nouvelles fonctions, ce qui n'est pas l'objectif de leur approche. De plus, le pouvoir prédictif de leur approche augmente rapidement dès le début du projet, pour atteindre ses meilleures performances lors de la phase de maintenance. Plus le modèle peut apprendre de données à partir de l'historique, plus le nombre et la qualité des suggestions sont élevés.

2.2.3 Duplication de code

Göde et Koschke [21] ont analysé la façon dont évoluent les clones de code pendant leur cycle de vie en calculant leur fréquence de changement et le risque d'inconsistances involontaires qui en découlent dans trois systèmes matures. La duplication de fragments de code est une pratique qui exhibe plusieurs problèmes comme une augmentation de la taille du code source, un effort plus important pour effectuer un changement et un risque exacerbé d'inconsistances dues à une propagation incomplète des changements. Cependant, tous les clones ne sont pas forcément nuisibles à l'évolution d'un système. C'est par exemple le cas des clones qui ne sont pas modifiés pendant leur cycle de vie, ce qui limite fortement leur impact. Il n'est donc pas toujours pertinent d'utiliser des ressources pour gérer ce type de clones.
L'objectif était d'estimer la menace potentielle des différents clones sur l'évolution des systèmes, afin de pouvoir orienter l'effort des développeurs vers les candidats effectivement nuisibles.

Leur étude explore l'historique des clones du système et se base sur une approche par jeton, c'est-à-dire que les fichiers du système sont traités comme des séquences de jetons (à la différence d'une approche basée sur le texte ou sur des arbres syntaxiques abstraits par exemple). Un fragment est défini comme une section continue de code source. Chaque fragment possède un

nombre quelconque d'ancêtres et zéro ou un descendant, qui sont respectivement ses occurrences dans la version précédente et son occurrence dans la version suivante du système. De plus, une paire de clones correspond à deux fragments similaires qui appartiennent à la même version du système.

Les auteurs distinguent trois types de clones, suivant leur degré de similarité :

— Type 1 : Les séquences de jetons des deux fragments sont identiques, les espaces et les commentaires ne sont pas pris en compte.

— Type 2 : Il s'agit d'une paire de clones de type 1, à la différence que la valeur des identificateurs et des littéraux n'est pas considérée.

— Type 3 : C'est une paire de clones de type 2 mais où certains jetons peuvent exister uniquement dans un des fragments.

Une classe de clones est définie de la même manière qu'une paire de clones, si ce n'est qu'elle regroupe deux fragments similaires ou plus.
Göde et Koschke définissent le graphe d'évolution des clones comme une représentation sous forme d'hypergraphe des fragments clonés et de leurs relations. Les fragments sont les noeuds, les relations entre les classes de clones sont les hyperarêtes qui connectent un ou plusieurs fragments d'une même version, et les liens de parenté sont les arêtes qui connectent deux fragments de deux versions successives.

La première partie de leur étude consiste à extraire les données d'évolution des clones depuis l'historique de chaque système. Ce processus est basé sur leur algorithme de détection incrémental de clones qui vise à détecter les clones dans les multiples versions des systèmes.

A partir de ces données, ils fournissent un outil pour générer le graphe d'évolution des clones. Les deux métriques suivantes y sont ensuite calculées :

1. La fréquence de changement des clones pendant leur cycle de vie, calculée comme étant le nombre de classes de clones marquées comme

changées pour chaque généalogie du graphe. Cette mesure vise à estimer les coûts additionnels induits par la propagation des changements des clones. Une absence de changement n'apporte donc aucun coût de propagation supplémentaire.

2. Le risque d'inconsistances involontaires lié aux clones. Cette métrique est calculée à travers une évaluation manuelle des changements des clones de manière à déterminer ceux qui n'ont pas été correctement propagés. De par l'effort requis pour ce type d'évaluation, les auteurs l'ont limitée à un seul système et uniquement au premier type de clone.

L'application de ces métriques montre que la majorité des clones ne provoque pas d'effort additionnel. Observer l'historique de l'évolution des clones permet donc d'apporter des informations importantes pour décider de leur pertinence. De même, ils ont observé que les changements appliqués aux clones sont souvent cohérents.

2.2.4 Coévolution du code de production et du code de test

Zaidman et al. [22] ont étudié la coévolution entre le code de production et le code de test en explorant le gestionnaire de version de trois systèmes logiciel. Le développement logiciel est une activité qui comprend la production d'une multitude d'artefacts. Étant donné sa position centrale dans le développement logiciel, l'artefact principalement considéré est typiquement le code source. Cependant, d'autres produits logiciel revêtent également une grande importance pour le développement d'un code de haute qualité (e.g., la documentation, les tests, les exigences etc.). Un défi inhérent à cette multidimensionnalité réside dans le fait que, de manière à assurer la cohérence entre les différents artefacts, la modification d'un artefact doit toujours être reflétée dans les autres artefacts liés [5]. En d'autres termes, il est important d'établir une *coévolution* entre l'ensemble des artefacts liés.

Le code de production et le code de test sont deux artefacts particulièrement dépendants l'un envers l'autre. En effet, il est de nos jours pratique courante d'accompagner l'écriture du code source par l'exécution de tests, et

ce tout au long du cycle de développement logiciel. Ceci de manière à repérer les erreurs potentielles dans le plus court délai possible. Certaines stratégies de développement préconisent même d'écrire certains tests avant d'en écrire le code source associé. L'analyse de ces deux artefacts a pour but d'identifier plusieurs scénarii de coévolution afin de permettre aux développeurs et aux managers d'être davantage conscients du processus de test pratiqué.

Les auteurs ont proposé trois vues complémentaires qui permettent de visualiser la façon dont évoluent le code de production et le code de test. L'utilisation de méthodes de visualisation permet d'interpréter plus aisément les larges volumes de données emmagasinés dans l'historique d'un système logiciel. Ces vues sont les suivantes :

1. Vue sur l'historique de changement : Cette vue permet de visualiser la façon dont les commits du code de production et du code de test sont réalisés au fil du temps. Elle permet entre autre de déterminer si les modules du code de production possèdent un test associé et si les deux types de code sont ajoutés ou modifiés au même moment.

2. Vue sur l'historique de croissance : Cette deuxième vue montre la croissance relative du code de production et du code de test dans le temps. Elle vise à identifier les patterns indiquant un développement synchrone ou asynchrone du code de production et du code de test, à observer si l'activité d'écriture de test augmente pendant la période qui précède la publication d'une version majeure, et à observer les manifestations d'un développement piloté par les tests.

3. Vue sur l'évolution de la couverture de test : Enfin, cette vue permet d'évaluer la couverture de test des composants logiciel, ce qui permet de donner une idée de la qualité des tests. En particulier, elle permet d'observer la quantité de code de test écrite pour couvrir un niveau particulier de code de production, et d'observer si une fluctuation du nombre de lignes de code de test est reflétée au niveau du degré de la couverture de test du système.

Zaidman et al. ont validé l'utilisation de ces vues sur deux systèmes *open source* et un système industriel. Ils ont observé que la manière dont était conduite la coévolution entre les deux types de code (i.e., synchrone ou par phases) était déterminée par l'approche de développement suivie par les développeurs. De plus, aucune augmentation de l'activité de test avant la publication d'une version majeure n'a été remarquée. Les auteurs attribuent cela à la nature open-source des deux premiers systèmes, et aux pratiques d'intégration continue appliquées dans le système industriel. Néanmoins leurs techniques de visualisation ont permis d'identifier des périodes de test intenses dans l'historique de développement. Ils ont également pu identifier la manifestation d'un développement piloté par les tests dans deux des trois systèmes, grâce au commit simultané du code de production et du code de test. Finalement, l'augmentation de la couverture de test semble entraîner une augmentation du nombre de lignes de code de test. L'exploration de l'historique des trois systèmes a donc permis aux auteurs de déduire des informations pertinentes sur les processus de test pratiqués.

2.3 Techniques d'analyse pour l'évolution de systèmes data-intensive

Un grand nombre d'études empiriques se sont consacrées à l'évolution des systèmes logiciels. Toutefois, peu d'entre elles se sont concentrées sur l'analyse de l'utilisation des base de données dans des systèmes logiciels *data-intensive*. La base de données est pourtant un artefact central dans ce type de systèmes, au vu des fortes interactions qui existent entre cette dernière et les programmes d'application.

Dans ces systèmes, il est essentiel de maintenir la cohérence entre trois composants [23] :

1. Le schéma de la base de données, qui doit respecter le modèle de données de la technologie de gestion de bases de données utilisée.

2. Les données de la base de données, qui doivent être structurées selon la structure définie par le schéma.

3. Le programme d'application, dont les requêtes d'accès doivent respecter la structure de données du schéma, et l'interface de programmation utilisée pour accéder aux données.

De ce fait, la coévolution entre le code source et la base de données est très importante pour maintenir et faire évoluer des systèmes *data-intensive*.

Cette section présente dès lors plusieurs études traitant de l'évolution de systèmes *data-intensive* et des défis liés aux besoins de coévolution exposés ci-dessus.

2.3.1 Migration de systèmes d'information

Lors des phases de conception, de développement et de maintenance d'un logiciel dit "*data-intensive*", la complexité et la volumétrie des données vont être croissantes dans le temps. L'évolution du logiciel va donc impliquer une évolution progressive de la manière dont les données sont stockées et gérées dans la/les base(s) de données. Par exemple, il n'est pas rare que des tables y soient ajoutées, supprimées ou modifiées. Ce phénomène propre à l'évolution de toute base de données possède donc évidemment des répercussions au niveau des requêtes d'accès présentes dans le logiciel : leur cohérence et leur validité doivent être maintenues à tout moment en fonction de l'évolution de la base de données.

Dans la littérature, plusieurs contributions sont à noter concernant ce phénomène. Hainaut et al. [24] évoquent la migration de plate-forme d'applications business très volumineuses construites autour d'une base de données. La problématique concerne d'une part la conversion de la base de données vers un nouveau paradigme de gestion des données (aussi bien au niveau du schéma qu'au niveau des données en elle-même), et d'autre part l'adaptation des programmes d'application vers le schéma de la base de données migrée et le système cible de gestion des données. Une structure de référence à deux dimensions est alors proposée afin de solutionner la problématique en question. Cette dernière identifie six stratégies différentes afin d'opérer la migration, où la première dimension concerne la conversion de base de données et la seconde la conversion de programme.

La cohérence entre l'évolution des requêtes d'accès et du schéma de la base de données symbolise donc le noeud de cette problématique. Afin d'assurer la cohérence future du système, l'analyse de l'évolution historique des requêtes d'accès d'une part et du schéma de la base de données d'autre part est l'élément central à traiter. Par conséquent, cela induit un besoin tangible de la création et l'utilisation de technologies pouvant traiter cet aspect d'évolution parallèle, la contribution apportée dans le mémoire répondant à l'amont du problème via son analyse des dépôts logiciel.

2.3.2 Évolution du schéma d'une base de données

L'évolution du schéma de base de donnée est un sujet qui a été abondamment étudié dans la littérature [25]. L'étude de Cleve et al. [26] est particulièrement intéressante puisqu'elle porte sur l'analyse de très larges schémas de bases de données relationnelles. Les auteurs ont présenté une approche générique qui utilise le gestionnaire de version de façon à produire une vue globale et intégrée de l'historique du schéma d'un système *data-intensive*.

Le schéma d'une base de donnée relationnelle est une structure importante puisqu'il contient une description des concepts spécifiques au domaine d'application du système. Selon [27], plus de 50% des structures et des constructions d'un schéma peuvent être implicites (e.g., les clés étrangères), ce qui signifie que leur présence n'est pas explicitement déclarée dans le système de gestion de base de données. En l'absence de documentation disponible, il est nécessaire de se baser sur d'autres artefacts pour reconstruire les connaissances associées.

Cleve et al. [26] ont analysé l'historique de l'évolution de bases de données *legacy* comme source supplémentaire d'information pour aider le processus de rétro-ingénierie de base de données. Ils ont appliqué leur approche sur un système *data-intensive* populaire de dossier médical nommé *OSCAR*. La dernière version de son schéma contenait plus de 400 tables, pour une période de développement de dix ans.

Ils ont extrait et comparé les versions successives du schéma logique de la base de données à partir des fichiers SQL collectés dans le gestionnaire de version. Le *schéma historique global* est ensuite construit incrémentale-

ment. Celui-ci est une représentation visuelle et explorable de l'évolution du schéma de base de données au cours du temps. Il contient tous les objets ayant un jour existé dans l'historique de schéma, i.e., les tables, les colonnes et les contraintes. De plus, chaque objet est annoté avec des méta-informations liées à leur cycle de vie (e.g., la date de création de l'objet, la liste des versions du schéma où l'objet est présent etc.), ce qui permet une analyse détaillée de chaque objet. Le *schéma historique global* peut ensuite être interrogé afin d'en extraire des informations utiles sur l'évolution de la base de données. Notamment d'identifier les tables les plus stables avant la migration des données vers un autre type de base de données. De même, il peut être intéressant d'identifier des fragments de schéma qui ne sont plus utilisés, mais qui sont toujours présents pour diverses raisons.

La Figure 2.1 montre un exemple de l'évolution d'un schéma de base de données sur trois versions successives. Le schéma S1 correspond à la version la plus vieille, tandis que le schéma S3 est le plus récent. Plusieurs changements ont eu lieu entre la version 1 et la version 2 du schéma. La colonne A2 a été supprimée de même que toute la table C, tandis que la colonne B2 et la table D ont été ajoutées. A la version 3, la table D est restée inchangée, la table B a disparu et la table C est réapparue. En effet, celle-ci était déjà présente dans la version 1 mais a été supprimée dans la version 2, avant de réapparaître dans la version 3. Ce phénomène illustre les différentes "vies" qu'un objet du schéma peut avoir. La Figure 2.2 illustre le schéma historique global dérivé à partir de cet historique d'évolution. Tous les objets ayant existé dans le cycle de vie de la base de données y sont représentés.

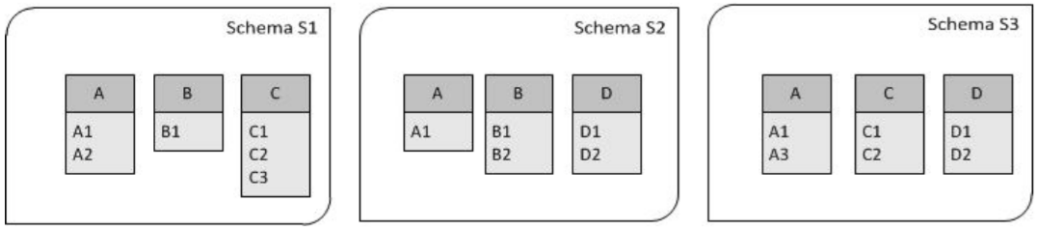

FIGURE 2.1 – Exemple d'évolution du schéma, proposé par Cleve et al. [26]

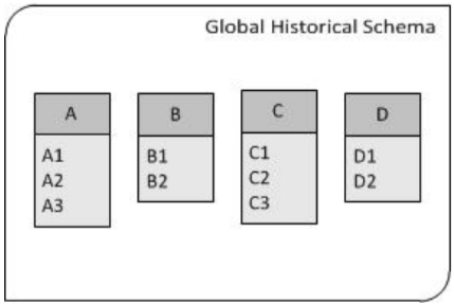

FIGURE 2.2 – Dérivation du *schéma historique global* à partir de la Figure 2.1 (Cleve et al.)

L'analyse de Cleve et al. considérait uniquement le schéma de bases de données. Cependant, un autre aspect essentiel à traiter pour maintenir une cohérence entre la base de donnée et le programme d'application est d'analyser en amont l'évolution des requêtes d'accès vers la base de données. Pour ce faire, la première étape est de localiser et d'extraire ces requêtes. Le point suivant aborde cette problématique.

2.3.3 Localisation et extraction des requêtes d'accès

Meurice et al. [28] ont proposé une technique d'analyse statique qui permet aux développeurs de localiser automatiquement les emplacements du code source où sont exécutées les requêtes d'accès vers une base de données relationnelle, et d'extraire l'ensemble des requêtes SQL potentiellement exécutables à ces localisations. Identifier et analyser les requêtes de base de données exécutées par un programme d'application peut être utile dans de nombreux scénarii, que ce soit pour des besoins de rétro-ingénierie de base de données, la migration d'une plate-forme de base de données ou une analyse d'impact des changements du schéma d'une base de données.

Le lien entre le programme d'application et la base de données d'un système *data-intensive* est réalisé par le biais d'une interface de programmation, qui définit la manière dont les deux composants interagissent. Dans les systèmes modernes tels que ceux utilisant Java, l'accès vers les bases de données est de plus en plus souvent effectué d'une manière dynamique : les requêtes sont construites à la volée, lors de l'exécution du programme. Dans le plus simple des cas celles-ci résultent d'une concaténation de String, mais

elles peuvent aussi être générées par des frameworks proposant un mapping objet-relationnel (ORM). Le but de ce type de framework est de convertir les données utilisées par un programme orienté objet (e.g., classes, attributs) en données utilisées par une base de données relationnelle (e.g., tables, colonnes), ce qui permet alors d'effectuer des opérations de persistance directement sur des données objet. Cette dynamicité accrue a pour conséquence de compliquer davantage la reconstruction et la récupération des requêtes d'accès exécutées par un programme.

Les auteurs ont proposé une approche qui considère trois technologies populaires d'accès à une base données relationnelle utilisées dans les systèmes Java : JDBC, Hibernate et JPA. La première est l'interface de programmation Java standard, qui fourni un bas niveau d'abstraction pour connecter un programme d'application à une base de données relationnelle, tandis qu'Hibernate et JPA sont des solutions avec un plus haut niveau d'abstraction, la première étant un framework ORM, et l'autre la spécification d'une interface de programmation Java qui fournit également un mapping objet-relationnel. La Figure 2.3 présente une vue d'ensemble de cette approche.

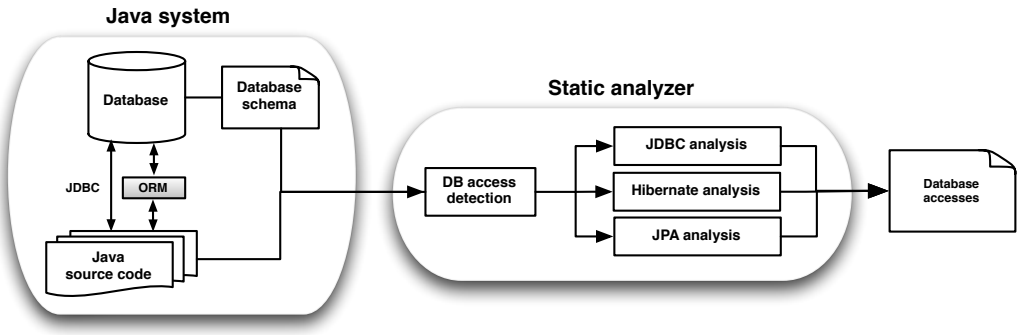

FIGURE 2.3 – Vue d'ensemble de l'approche proposée par Meurice et al. [28]

La première étape consiste en l'extraction du graphe d'appel des méthodes du système Java. En effet, les accès BD sont typiquement construits en utilisant certains paramètres d'entrée de la méthode englobante. Il est dès lors nécessaire d'explorer le graphe d'appels pour déterminer les différentes valeurs possibles de ces paramètres. Pour ce faire, les auteurs se sont basés sur une analyse inter-procédurale du programme.

L'étape suivante se charge de détecter l'emplacement dans le code source de l'entièreté des requêtes d'accès JDBC, Hibernate et JPA du système. Leur analyseur Java s'appuie sur un arbre syntaxique abstrait de manière à parcourir l'ensemble des noeuds et des expressions Java du système. Chaque accès réalisé depuis les méthodes des technologies considérées est détecté et envoyé à l'analyse correspondante dans la phase qui suit.

Enfin, la dernière étape s'occupe de l'analyse des accès détectés. Dans le cas des accès JDBC, le parser extrait simplement la chaîne de caractères correspondant à la requête SQL, puis identifie les tables et les colonnes accédées par celle-ci en s'appuyant sur le schéma de la base de donnés. L'analyse d'Hibernate est plus sophistiquée. Tout comme JDBC, Hibernate propose des méthodes Java pour exécuter des requêtes SQL ou HQL. Pour parser les requêtes HQL, celles-ci sont d'abord transformées en requêtes SQL en invoquant le compilateur interne d'Hibernate avec le même contexte qui serait utilisé à l'exécution. Hibernate propose également un ensemble de méthodes qui accèdent à la base de données en opérant sur les instances Java des classes d'entité mappées. Pour traiter ces opérations, les auteurs identifient d'abord la classe d'entité de l'objet en entrée, puis détectent les objets de la base de données correspondant à ce mapping. L'analyse des accès JPA est similaire à celle d'Hibernate.

Le résultat de ce processus correspond à la localisation de l'ensemble des accès vers la base de données, et aux objets BD (i.e., les tables, les colonnes) accédés pour chacun d'eux.

Ils ont ensuite validé leur approche sur les trois systèmes *open-source data-intensive OSCAR*, *OpenMRS* et *Broadleaf Commerce*. Ils ont pu extraire les requêtes pour 71.5% - 99% des accès BD avec 87.9% - 100% de requêtes valides. Il s'agit donc d'une approche qui se montre applicable en pratique à de larges projets Java.

L'identification et l'extraction des requêtes d'accès présentes dans un système logiciel permet une analyse détaillée de leur cycle de vie. Les requêtes d'accès n'évoluent toutefois pas de manière isolée mais bien en conjonction avec d'autre artefacts. Cette évolution est notamment ponctuée au rythme des changements appliqués au schéma de la base de données. Le point sui-

vant aborde le défi lié à la gestion de la coévolution entre un système logiciel et un système de données, et des effets des différentes technologies d'accès sur celle-ci. Il s'agit d'un thème primordial pour faire évoluer d'une manière gracieuse et durable des systèmes "*data-intensive*".

2.3.4 Technologies d'accès et coévolution

Dans [29], Meurice et al. ont étudié d'une part la coévolution entre différentes technologies d'accès utilisées dans le langage Java, et d'autre part la coévolution du code source et du schéma de bases de données relationnelle au sein de trois systèmes Java *data-intensive*. Il n'est pas rare que plusieurs technologies d'accès soient utilisées simultanément au sein d'un même système logiciel. Dans ce contexte, il peut être intéressant pour les développeurs et les managers d'avoir des informations sur la façon dont évoluent et coexistent ces différentes technologies. Les auteurs ont proposé une vue d'ensemble historique de l'utilisation des technologies d'accès à travers une étude empirique à grande échelle de plusieurs milliers de projets Java. Grâce à celle-ci, il est possible d'identifier les technologies les plus durables, celles qui tendent à être supplantées rapidement, ou encore les technologies utilisées de manière complémentaire.

Leur analyse considérait quatre technologies : JDBC, Hibernate, JPA et Spring. Cette dernière n'est pas une solution de persistance en elle-même, mais un framework permettant une intégration aisée de plusieurs technologies d'accès à travers la mise en oeuvre de sa couche de persistance. Ils ont observé à travers leur étude empirique que, malgré l'utilisation encore très répandue de technologies d'accès bas niveau comme JDBC, les interfaces de programmation plus évoluées comme JPA et Hibernate continuent de gagner en popularité. De plus, presque 40% des systèmes analysés utilisaient au moins deux technologies d'accès différentes pendant la période observée, la majorité du temps de manière simultanée. Par ailleurs, les nouvelles technologies sont généralement plus souvent et plus rapidement introduites (et moins vite remplacées) dans des larges systèmes que dans des systèmes de plus petite taille. Enfin, ils ont observé qu'Hibernate a tendance a être rapidement substitué ou complimenté par une autre technologie.

Les auteurs ont complété cette analyse globale par une analyse plus spécifique de trois larges systèmes Java *open-source* et *data-intensive* (i.e., *OSCAR*, *OpenMRS* et *Broadleaf Commerce*) utilisant les différentes technologies d'accès étudiées dans l'analyse précédente. Cette seconde analyse permet de déterminer la façon dont les systèmes évoluent au cours du temps, dont se déroule la coévolution entre la base de données et le code source (e.g., pour identifier la présence d'inconsistances), et les raisons qui gouvernent les changements de technologies d'accès.

A partir de la localisation des requêtes d'accès dans le code source pour les versions de chaque système (obtenue grâce à la méthode développée dans [28]), Meurice et al. ont construit un schéma conceptuel générique contenant tous les artefacts pouvant être analysés historiquement dans un système *data-intensive*. Dans ce schéma, à chaque objet du système, que ce soit un objet du code (e.g., une méthode, une classe), un objet du schéma (e.g., une table, une colonne), un mapping ORM, ou un accès à la base de données, est assigné un numéro de version. Cela permet d'analyser l'évolution de chaque composant du programme et des données au cours du temps.

Sur base de ce schéma, ils ont ensuite calculé plusieurs métriques de manière à étudier les caractéristiques d'évolution spécifiques à chaque système. Leurs observations montrent que certaines tables sont accédées par plusieurs technologies d'accès au sein d'un même système, signe d'une migration incomplète de technologie. D'autre part, les développeurs semblent éviter de faire évoluer les tables du schéma, de crainte d'invalider les requêtes d'accès associées. En effet, une partie significative des tables et des colonnes des schémas observés ne sont plus accédées par aucune requête des programmes d'application, mais continuent néanmoins d'exister dans le schéma. A la place, les développeurs semblent préférer l'ajout d'une nouvelle table contenant les données dupliquées, pour ensuite mettre à jour incrémentalement les programmes d'application de manière à utiliser la nouvelle table en lieu et place de l'ancienne.

L'approche présentée ci-dessus constitue une première étape vers un système de recommandation permettant d'assister les développeurs dans des tâches de coévolution entre le programme d'application et la base de données. Il s'agit précisément du sujet abordé dans la suite des recherches de

Meurice et al., présentée au point suivant.

2.3.5 Détection et prévention d'incohérences

Meurice et al. [30] ont poursuivi leurs recherches en développant un outil permettant aux développeurs de détecter et de prévenir les incohérences dans le programme d'application lors de l'évolution du schéma de base de données. L'utilisation des requêtes d'accès construites dynamiquement est de plus en plus populaire dans les systèmes logiciel *data-intensive*. De par cette dynamicité et l'abstraction supplémentaire induite par celle-ci, il est moins évident pour les développeurs d'identifier et de localiser les éléments du programme d'application impactés par une modification du schéma de base de données. Il s'agit pourtant d'une tâche essentielle pour conserver la cohérence entre ces deux artefacts, capitale pour le bon fonctionnement du système logiciel.

La première étape de leur approche visait à identifier la présence d'incohérences dues à des précédents changements du schéma dans les trois systèmes considérés, qui ne sont autres que *OSCAR*, *OpenMRS* et *BroadLeaf Commerce*. Cette démarche repose sur des outils et méthodes développés dans leurs travaux précédents afin de construire pour chaque système un modèle générique contenant l'ensemble de leurs données historiques [28, 29]. Ce modèle est composé de quatre parties principales : l'historique du code source, l'historique du schéma, l'historique des mappings ORM et l'historique des requêtes d'accès. A partir de ces données, les auteurs ont évalué l'effort qui était requis dans le passé pour adapter le code source en réaction à l'évolution du schéma de la base de données. Quatre types de changement ont été considérés, à savoir : la suppression d'une table, le renommage d'une table, la suppression d'une colonne et le renommage d'une colonne. Plusieurs métriques ont ensuite été utilisées pour estimer les efforts et les difficultés qui surviennent lorsque les développeurs adaptent *manuellement* le code source suite à des changements au niveau du schéma de la base de données. Ils ont notamment observé que certaines modifications du schéma peuvent nécessiter plusieurs versions avant d'être complètement propagées au niveau du code source. D'autres changements du schéma n'ont d'ailleurs jamais été totalement propagés, ce qui est susceptible d'introduire des défauts dans le code. De surcroît, il arrivait même dans certains cas aux développeurs de continuer à créer des accès vers des objets du schéma supprimés ou

renommés. Ces résultats soulignent donc la nécessité d'un outil qui assiste les développeurs dans des tâches de coévolution, loin d'être toujours triviales.

Pour faire face à ces problèmes, les auteurs ont proposé une approche automatique permettant aux développeurs de simuler des changements sur le schéma de la base de données de façon à déterminer les objets impactés dans le code source. Leur approche prend en entrée une version particulière du système (e.g., la version courante du système) et un changement hypothétique du schéma de la base de données (e.g., la suppression d'une table t). Le résultat est une liste de recommandations indiquant l'endroit précis et la façon dont les modifications du schéma données en entrée devraient être propagées dans le programme d'application (e.g., *Vous avez besoin de supprimer le mapping Hibernate défini entre la table t et le classe Java c à cette localisation.*). L'objectif est d'assurer la préservation de la cohérence du système au cours du temps lors de l'évolution du schéma.

Ils ont évalué la précision des recommandations suggérées en vérifiant chacune d'elles manuellement au sein de l'historique d'évolution des trois systèmes considérés. Leurs résultats sont très prometteurs, puisque leur approche a atteint un score de 99% de recommandations correctes lorsqu'elle est appliquée à un sous-ensemble aléatoire et significatif de modifications du schéma de la base de données.

2.3.6 Documentation du code source lié à la base de données

Maintenir le code source pendant des tâches de coévolution avec la base de données peut se révéler être une tâche difficile et périlleuse pour les développeurs. Cela nécessite une compréhension des opérations liées aux accès BD, des méthodes du code source qui exposent ces accès, des contraintes imposées par le schéma, des contraintes implicites du schéma (e.g., quand le schéma n'utilise pas de clé étrangère), etc. Dans ce contexte, il est capital d'avoir à disposition une documentation complète et à jour du système. Cependant, documenter manuellement les opérations liées à la base de données dans un environnement en constante évolution est un problème complexe et chronophage.

Linares-Vásquez et al. [31] ont présenté une approche visant à générer de manière automatique une documentation en langage naturel continûment à jour, qui décrit les opérations de la base de données et les contraintes du schéma imposées sur ces opérations au sein des méthodes du code source. La première étape de leur approche vise à détecter les méthodes qui exécutent des instructions SQL au moyen d'une analyse statique du code source. Une fois ces méthodes localisées, l'étape suivante vise à propager les contraintes induites par le schéma de la base de données à travers la chaîne d'appels des méthodes qui implémentent les opérations SQL. En effet, le schéma de la base de données contient un ensemble de contraintes qui doivent être satisfaites lors d'opérations sur la base de données, i.e., lors d'insertions, de mises à jour, et de suppressions dans la BD. Par exemple, la suppression de la valeur d'une colonne considérée comme une clé étrangère dans d'autres tables ne peut pas être effectuée si d'autres tables référencent cette même colonne. Finalement, à partir de ces contraintes, une description en langage naturel est automatiquement générée pour chaque méthode de la chaîne d'appels au moyen de templates prédéfinis.

Ils ont évalué leur approche dans une enquête impliquant 52 participants qui avait pour but d'analyser la documentation générée au sein des méthodes de 5 systèmes *open-source data-intensive*. De plus, ils ont également validé les descriptions générées en interrogeant les développeurs de deux systèmes commerciaux *data-intensive*. Leurs résultats montrent que les descriptions générées par leur outil sont utiles pour améliorer la compréhension des usages et des contraintes de la base de données, notamment pendant des tâches de maintenance.

2.4 Contribution de ce mémoire

Notre travail poursuit les recherches de Meurice et al. présentées précédemment en proposant une méthode outillée d'analyse de l'évolution historique des requêtes d'accès au sein de systèmes *data-intensive*. Comparée aux études présentées ci-dessus, notre approche se concentre exclusivement sur l'analyse des requêtes d'accès à la base de données, et ce, de manière à décrire et prédire leur comportement au cours de leur cycle de vie. Cette méthode innovante s'appuie sur des techniques d'analyse d'origine et d'extraction de requêtes pour reconstruire le cycle de vie des requêtes d'accès

depuis les données extraites de l'historique de développement des systèmes. Les points suivants présentent les motivations et les objectifs de notre approche selon les perspectives différentes mais complémentaires du scientifique et du développeur.

2.4.1 Motivation théorique

Du point de vue scientifique, l'objectif est d'étudier la "vie" d'une requête d'accès. Ce terme se réfère à l'entièreté du cycle d'existence d'une requête, démarrant lors de sa création et se terminant lors de sa suppression définitive. Entre la création et la suppression définitive, son évolution doit donc être non seulement tracée, mais également analysée.

Cette analyse est composée de 2 angles différents, se complétant l'un et l'autre :

1. **Analyse intrinsèque :**

   La requête d'accès est analysée durant l'entièreté de son cycle d'existence, afin de déterminer en détails quelles modifications elle a subies au cours du temps. Cette analyse s'effectue indépendemment des autres requêtes présentes.

2. **Analyse globale :**

   Une fois l'analyse intrinsèque effectuée pour chaque requête, l'analyse globale consiste à produire une analyse statistique concernant l'ensemble des requêtes présentes dans le logiciel. Ces statistiques permettent de mettre en évidence la présence de phénomènes particuliers dans l'historique du système, tels que le temps de vie moyen et le nombre de modifications subies en moyenne entre deux périodes de temps succinctes.

Finalement, cette analyse à deux niveaux permet donc au scientifique de répondre à plusieurs objectifs de plus haut niveau, tels qu'énoncés à présent :

A. Découvrir des tendances/patterns/phénomènes particuliers concernant l'utilisation des différentes technologies d'accès à la base de données au cours du temps.

B. Analyser la cohérence entre l'évolution de la base de données et l'évolution des accès.

C. Déceler la présence de migration d'une technologie vers une autre, et analyser les impacts sur l'architecture du code.

L'analyse de l'évolution historique des requêtes d'accès se justifie par conséquent par la volonté du scientifique de répondre à l'un de ces objectifs, ou tout autre objectif possédant un niveau de granularité similaire.

2.4.2 Motivation pratique

Du point de vue du développeur, l'objectif intéressant est de pouvoir déterminer pourquoi une requête d'accès a dû être modifiée au cours du temps. En effet, une requête qui a été modifiée de très nombreuses fois (sur une certaine période de temps donnée) peut être synonyme d'un symptôme problématique qui était jusque là inconnu au développeur. Par conséquent, l'analyse de l'évolution des requêtes constitue une aide non négligeable dans la recherche de zones sensibles aux erreurs dans le logiciel.

Selon la loi de Pareto appliquée à l'étude des bogues dans les versions historiques d'un logiciel [34], 80% des bogues produits et visibles (c'est-à-dire ayant mené à une erreur notifiée) au niveau du code source sont répartis uniquement dans 20% des modules du système. Cette logique induit donc que certains modules sont plus sensibles que d'autres aux erreurs, et il est dès lors fort probable qu'un couplage temporel entre deux modules soit caché. Ce couplage temporel peut être repéré selon [33] en utilisant notamment une méthode consistant à repérer les modules ayant une fréquence de modification élevée dans un certain laps de temps, en recoupant cette information selon les modules qui ont été modifiés souvent ensemble durant un même commit.

Finalement, cette méthode était appliquée sur des fichiers complets, mais l'analyse de l'évolution des requêtes peut permettre de l'appliquer à présent également sur ces dernières de manière plus précise. Ainsi, le développeur possède alors des informations sur l'identification des requêtes dites plus sensibles aux erreurs.

Chapitre 3

Concepts préliminaires

Ce chapitre a pour objectif d'apporter une vue précise concernant certains concepts fréquemment utilisés dans les chapitres suivants. Il s'agit donc de concepts centraux vis-à-vis de la méthode que nous proposons d'établir dans le Chapitre 4. Par conséquent, cette lecture préliminaire permet de situer des termes dont la mention sera plus brève par la suite.

3.1 Mapping objet-relationnel

Le mapping objet-relationnel est une technique de programmation qui permet de donner l'illusion au programme qu'une base de données orientée objet est utilisée à la place d'une base de données relationnelle initialement. Il s'agit donc d'une technique utilisée dans le cadre des langages de programmation orientés-objet. Durant la suite du document, cette technique sera référée via l'utilisation de l'acronyme *ORM*, signifiant "*Object-Relational Mapping*".

La motivation de son utilisation réside dans le fait que les objets qui sont manipulés et stockés de manière persistante sont généralement des objets qui comprennent une structure de données qui n'est pas atomique. Dès lors, puisque des Systèmes de Gestion de Base de Données relationnelle peuvent manipuler et stocker uniquement des valeurs atomiques (comme des entiers ou des chaînes de caractères) organisées dans des tables, la persistance de ces structures non-atomiques demande au programmeur de les convertir au préalable en valeurs atomiques avant leur stockage. Par conséquent, le mapping objet-relationnel se voit utile afin d'automatiser ce processus à la place

du programmeur, et de le rendre également invisible et implicite pour ce dernier. Typiquement, un type d'objet peut représenter alors une entité à faire persister et qui se réfère à une table particulière au final.

Par exemple et pour illustrer ces propos, prenons le cas de la Figure 3.1 où la persistance et la suppression d'objets est effectuée sur base d'utilisation d'Hibernate. Hibernate est une libraire de mapping objet-relationnel pour le langage Java, qui permet de faire la correspondance entre des classes Java et des tables de la base de données relationnelle. Dans le cas présent de cette Figure 3.1, la ligne 12 permet de faire persister automatiquement l'instance de la classe Obs qui est censée représenter des Observations recueillies chez un patient. Ces Observations contenant des informations non-atomiques telles que des Personnes, des Médicaments et autres, il apparaît dès lors ici que leur persistance s'effectue de manière beaucoup plus concise que si le programmeur avait dû décomposer toutes ces informations de manière atomique. La ligne 16 présente le phénomène similaire pour la suppression de l'entité.

```
1  public void setSessionFactory(SessionFactory sessionFactory) {
2          this.sessionFactory = sessionFactory;
3  }
4
5  public void createObs(Obs obs) throws DAOException {
6          if (obs.getCreator() == null)
7                  obs.setCreator(Context.getAuthenticatedUser());
8
9          if (obs.getDateCreated() == null)
10                 obs.setDateCreated(new Date());
11
12         sessionFactory.getCurrentSession().persist(obs);
13 }
14
15 public void deleteObs(Obs obs) throws DAOException {
16         sessionFactory.getCurrentSession().delete(obs);
17 }
```

FIGURE 3.1 – Illustration d'opérations sur une instance de classe avec Hibernate

Hibernate fournit également un langage inspiré du SQL pour la création de requêtes, appelé Hibernate Query Language (abrégé par *HQL*). Il s'agit en fait d'un langage similaire à SQL à l'exception du fait qu'il permet d'écrire les requêtes en réutilisant les mappings de classe définis préalablement. La Figure 3.2 présente la manière dont se construisent donc ces requêtes, en reprenant l'exemple précédent des Observations. La requête s'exécute finalement à la ligne 15.

```
1  public List<Object[]> getNumericAnswersForConcept(Concept answer, Boolean
       sortByValue, Integer personType) {
2
3          String sort = "obs.obsDatetime desc";
4          if (sortByValue)
5                  sort = "obs.valueNumeric asc";
6
7          String sql = "";
8          sql += "select obs.obsId, obs.obsDatetime, obs.valueNumeric ";
9          sql += getHqlPersonModifier(personType, "obs.concept = :c and obs.
               valueNumeric is not null and obs.voided = false");
10         sql += " order by " + sort;
11
12         Query query = sessionFactory.getCurrentSession().createQuery(sql)
13                         .setParameter("c", answer);
14
15         return query.list();
16 }
```

FIGURE 3.2 – Exemple d'exécution de requête HQL

Ensuite, au delà d'Hibernate qui sera de nouveau évoqué dans les chapitres suivants, il faut également considérer la Java Persistance API (abrégé en *JPA*) qui est une spécification permettant de décrire la gestion de données relationnelles dans les applications. Tout comme Hibernate, le mapping entre les classes Java et les tables de la base de données permet d'effectuer les opérations sur les objets, leurs attributs et leurs relations plutôt que sur les tables et colonnes. Il existe alors un langage de requête orienté-objet, Java Persistence Query Language (i.e. *JPQL*), qui permet d'accéder aux entités de la base de données relationnelle et fait partie de la spécification de JPA. Son principe d'utilisation est fortement similaire à celui de HQL : la Figure 3.4 montre l'exécution d'une requête en JPQL (ligne 8) où la création de la requête à la ligne 5 trouve son origine dans le fichier de configuration de

mapping de la Figure 3.3 à partir de la ligne 7.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm http://java.
               sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
5
6
7          <named-query name="READ_USER_BY_USER_NAME" >
8                  <query>SELECT user FROM org.broadleafcommerce.profile.domain.
                       BroadleafUser user
9                  WHERE user.username = :username</query>
10         </named-query>
11
12     ...
13
14         <entity class="org.broadleafcommerce.profile.domain.BroadleafUser" name="
               user">
15         <table name="BLC_USER" >
16                 <unique-constraint>
17                         <column-name>USER_NAME</column-name>
18                 </unique-constraint>
19         </table>
20         <inheritance strategy="JOINED"/>
21         <attributes>
22             <id name="id">
23                 <column name="USER_ID"/>
24                 <generated-value/>
25             </id>
26             <basic name="username">
27                 <column name="USER_NAME" />
28             </basic>
29             ...
30
31         </attributes>
32     </entity>
33 </entity-mappings>
```

FIGURE 3.3 – Exemple de fichier de configuration de mapping de requêtes (JPA)

```
1  public User readUserByUsername(final String username) {
2      return (User) this.jpaTemplate.execute(new JpaCallback() {
3
4          public Object doInJpa(EntityManager em) throws PersistenceException {
5              Query query = em.createNamedQuery("READ_USER_BY_USER_NAME
                   ");
6              query.setParameter("username", username);
7              try {
8                  return query.getSingleResult();
9              } catch (NoResultException ne) {
10                 return null;
11             }
12         }
13     });
14 }
```

FIGURE 3.4 – Exemple d'exécution de requête JPQL

Finalement, l'utilisation d'une telle technique de mapping objet-relationenel possède des avantages et des désavantages. Le point positif principal est qu'elle permet de réduire au départ de manière drastique le nombre de lignes de code nécessaires pour gérer la persistance des objets vis-à-vis des manières plus traditionnelles existantes. Cependant, la contrepartie négative réside dans le fait qu'au cours du temps, cette technique a pour effet de rendre pauvre la qualité de conception de la base de données et de l'architecture du programme.

3.2 Analyse d'origine

L'analyse d'origine est le processus qui consiste à décider si une entité présente dans une version particulière d'un programme est une entité nouvellement introduite ou non, et ce en considérant une entité du code de manière très générale au départ. S'il s'agit d'une entité qui n'est pas nouvellement introduite, elle est dès lors considérée comme une version refactorisée, déplacée ou modifiée (d'une quelconque manière) d'une entité présente dans une version antérieure du programme, selon Godfrey et al. [32].

La motivation de son utilisation réside dans le fait que la réorganisation d'artefacts dans le code est un exercice très fréquent dans le cycle de vie du logiciel. Dès lors, l'historique de versions d'un système fait apparaître les

nouveaux artefacts répondant aux nouveaux besoins, mais ne fait pas apparaître leur origine et plus précisément la raison de leur évolution. Finalement, à défaut d'une documentation à jour et détaillée sur les raisons motivant les changements et leurs impacts dans le code, des outils doivent être utilisés pour extraire de l'information sur l'évolution du système après coup. Les outils en question reposent alors sur l'utilisation et l'implémentation concrète des méthodes heuristiques proposées par l'analyse d'origine.

Dans les travaux de Godfrey et al. [32], le champ d'application de l'analyse d'origine concerne plus spécifiquement des entités du logiciel telles que des fonctions ou des méthodes. Étant donné que ce type d'entités peut être identifié via ses relations d'appels entrants/sortants, deux versions successives d'une même fonction peuvent être identifiées comme une évolution effective si leurs relations d'appels sont très similaires. Cela permet notamment de pouvoir évaluer assez facilement, entre deux versions successives d'un système, quelles fonctions ont été fusionnées ou divisées en plusieurs nouvelles fonctions pour répondre aux besoins de nouvelles fonctionnalités (ou encore à des besoins de restructuration du code).

L'exemple sur la Figure 3.5 illustre le phénomène : dans le cas d'une ancienne fonction effectuant plusieurs tâches très diverses (i.e. la fonction F effectuant les tâches *out1* et *out2*), sa restructuration en de nouvelles fonctions séparées (i.e. les fonctions *G1* et *G2*) effectuant chacune une seule de ces tâches parait judicieuse pour maintenir la qualité du code. Par conséquent, si le développeur qui effectue cette restructuration omet de la documenter, il est difficile par après de pouvoir retrouver l'origine de ces nouvelles fonctions et les raisons de leur présence. Par conséquent, l'analyse d'origine permet de récupérer des informations enrichissantes dans l'évolution logicielle.

Finalement, l'analyse d'origine telle que proposée précédemment est bidirectionnelle. C'est-à-dire qu'elle peut être appliquée à la fois d'anciens artefacts vers de nouveaux, ou inversement. Cela permet de pouvoir à la fois analyser et récolter des informations sur les nouveaux artefacts créés et sur ceux qui ont été supprimés. L'objectif est de pouvoir déterminer si ils l'ont réellement été ou ont simplement fait l'objet d'une modification profonde, ce qui est potentiellement intéressant à connaître pour le développeur.

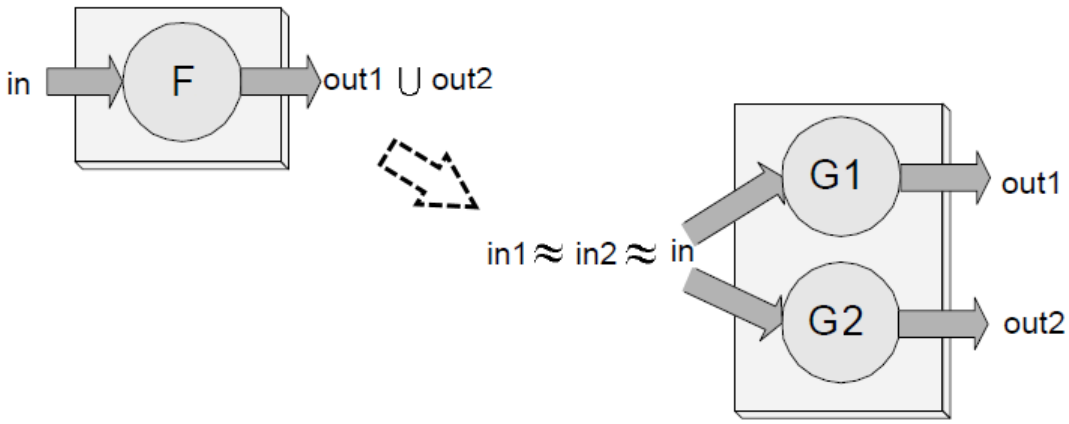

FIGURE 3.5 – Restructuration de fonction proposée par Godfrey et al. [32]

3.3 Approche du Mining Software Repositories

Trois grandes étapes sont typiquement suivies pour analyser l'historique d'évolution d'un système logiciel à partir de ses dépôts logiciels [14] :

1. Modélisation des données : Cette étape consiste en la création du modèle de données qui représente les données disponibles dans les artefacts d'un système et leurs relations. Plusieurs aspects du système logiciel peuvent être modélisés comme la dernière version du code source, l'historique de l'évolution des fichiers du système (à partir du gestionnaire de version), la documentation, les archives e-mail, l'historique d'exécution ou les rapports de bogue. Pour ne pas traiter un ensemble de données inutilement grand dans les phases en aval, il est particulièrement important de sélectionner uniquement les aspects qui peuvent adresser un ensemble spécifique de problèmes.

2. Extraction et traitement des données : Une fois le modèle défini, l'étape suivante est la création d'une instance de ce modèle. Les données vont d'abord être extraites des différentes sources considérées, avant d'être parsées et traitées. Ce processus inclut aussi la mise en correspondance des données des différents artefacts (e.g., pour lier un rapport de bogue à une entité du code source, pour associer différents noms utilisés par un même développeur au sein des artefacts), la reconstruction d'informations qui n'ont pas été enregistrées, et l'application d'autres techniques

comme le *data mining*.
Parmi ces données, certaines peuvent être plus difficiles à manipuler. C'est le cas des données non structurées, qui n'ont ni une structure claire ni une sémantique évidente (e.g., le texte brut dans les rapports de bogue ou les commentaires du code source) [15]. Dans certains cas il peut donc être nécessaire d'avoir recours à des heuristiques.

3. Analyse des données : La dernière étape consiste en l'analyse de ces données et en l'interprétation des patterns extraits afin de découvrir des faits intéressants sur le système logiciel. Plusieurs techniques et outils peuvent être utilisés lors de cette étape, les techniques de visualisation étant par exemple communément utilisées pour aider à l'interprétation des métriques obtenues.

Chapitre 4

Méthode d'analyse d'évolution des requêtes

Ce chapitre a pour objectif de présenter en détail le coeur de la méthode qui a été élaborée. Tout d'abord, la Section 4.1 et ses sous-sections dressent le cadre initial qui ont conduit à la création de la méthode, ainsi que la terminologie de l'évolution de requêtes afin de fixer quelques définitions pour le reste du chapitre. Dans ce cadre initial, un point important abordé est la combinaison entre les travaux de Meurice et al. sur l'analyse statique et l'extraction de requêtes, et une contribution personnelle quant à l'adaptation de l'analyse d'origine d'artefacts de manière spécifique aux requêtes. Le choix de cette combinaison et sa création représentent le principe premier de la méthode proposée.

Ensuite, la Section 4.2 présente le coeur de la méthode et son approche par heuristiques. Par conséquent, les différentes heuristiques permettant d'analyser les requêtes qui ont été extraites du code source sont explicitées. Dès lors, la méthode finale est élaborée sur base de la combinaison entre ces heuristiques et les travaux antérieurs de Meurice et al. .

Finalement, la Section 4.3 et ses sous-sections contribuent à rendre cette méthode outillée. Les choix d'implémentation ainsi que l'architecture choisie y sont brièvement discutés, ainsi que les fonctionnalités retrouvées.

4.1 Éléments de réflexion préliminaires

Afin de mettre en place et de détailler la méthode élaborée, il est nécessaire de définir au préalable les concepts sous-jacents. Dès lors, sont présentés dans cette section les différents éléments théoriques constituant la base de la réflexion amenant à la construction de la méthode proposée, ainsi que les motivations derrière l'emploi d'une telle méthode.

4.1.1 Cadre de travail initial

En amont de l'élaboration de notre méthode se situe un travail d'une importance majeure afin de poser les tenants et les aboutissants de nos recherches. En effet, les travaux de Meurice et al. présentés en détail dans la Section 2.3.3 et les sections suivantes constituent les prémisses des recherches du chapitre présent. Plus particulièrement, l'élément central est la technique d'analyse statique élaborée par Meurice et al. qui permet de localiser dans le code source où sont exécutées les requêtes d'accès à la base de données relationnelle, et de déterminer et d'extraire les requêtes SQL susceptibles d'être exécutées à ces endroits. Cette technique d'analyse constitue alors le premier élément dont nécessite la méthode établie dans ce chapitre. Le méta-modèle produit par la technique d'analyse, ainsi que la méthode de traduction de requêtes d'un langage vers SQL sont dès lors présentés.

Méta-modèle et données initiales

La technique d'analyse statique, appliquée sur différents cas d'étude représentant des historiques de systèmes découpés en versions successives, a permis de récolter des méta-informations sur ces requêtes. Dès lors, Meurice et al. ont créé un méta-modèle permettant de conserver ces informations pour pouvoir analyser plus tard l'historique de ces systèmes. Le méta-modèle en question comprend 4 grands modules différents :

A. Le module d'*historique du code*, qui représente les composants du programme tels que les fichiers, les classes, les méthodes et dont l'emplacement de leur définition est capturé via un couple de coordonnées, à savoir leur ligne de début et leur ligne de fin. (cf. Figure 4.1)

B. Le module d'*historique du schéma de la base de données*, qui contient les tables et colonnes. Pour ces dernières, des informations sur leur type sont également disponibles. (cf. Figure 4.2)

C. Le module d'*historique des accès à la base de données*, qui représente les accès qui apparaissent dans le programme sous forme d'appel de méthode à une méthode particulière contenue dans une API, une libraire ou un framework responsable de la persistance des données. (cf. Figure 4.3)

D. Le module d'*historique des mappings ORM*, qui contient les informations relatives aux accès qui se déroulent via un mapping objet-relationnel. Ces accès sont donc systématiquement des accès créés avec JPA ou Hibernate dans notre cas, les requêtes SQL fournissant un accès direct à des objets de la base de données n'étant pas concernées. (cf. Figure 4.4)

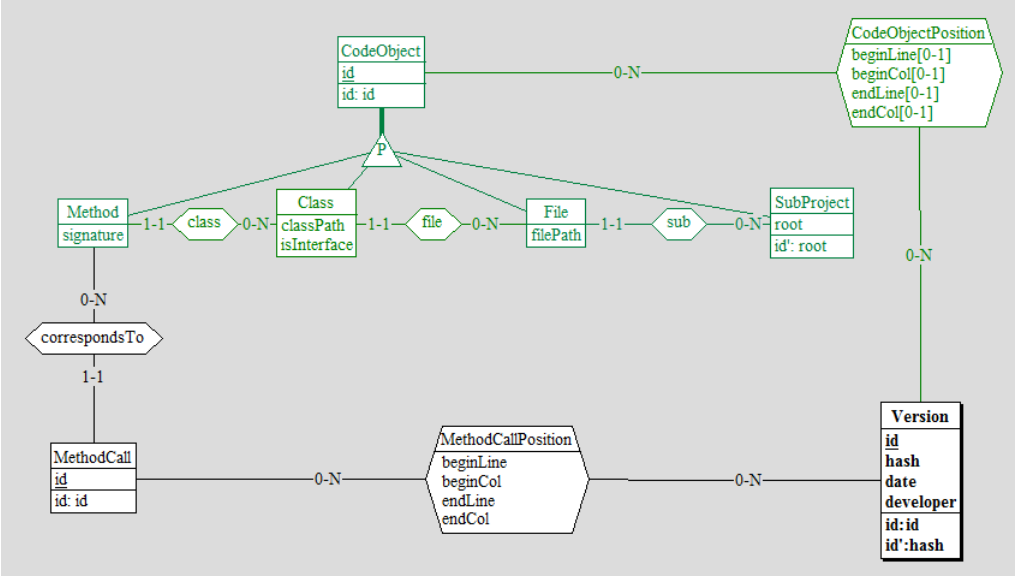

FIGURE 4.1 – Module d'historique du code (Meurice et al.)

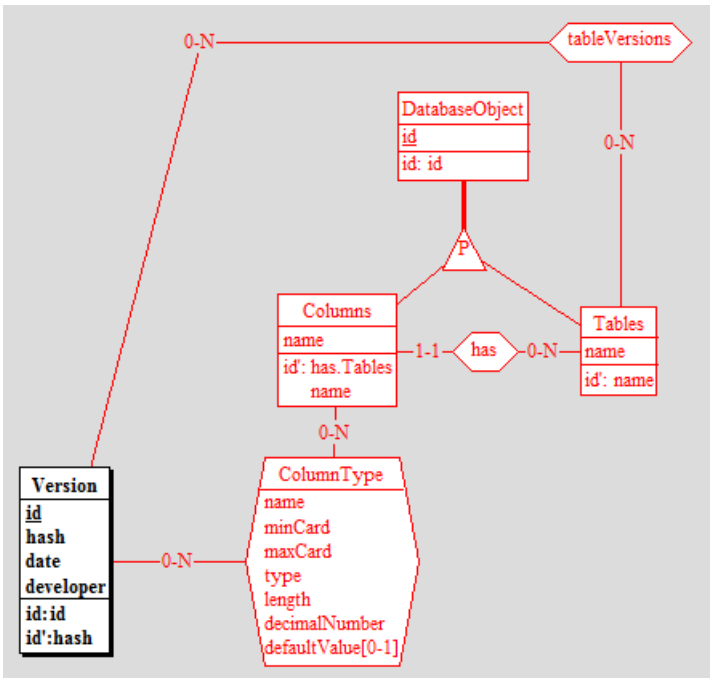

FIGURE 4.2 – Module d'historique du schéma de la base de données (Meurice et al.)

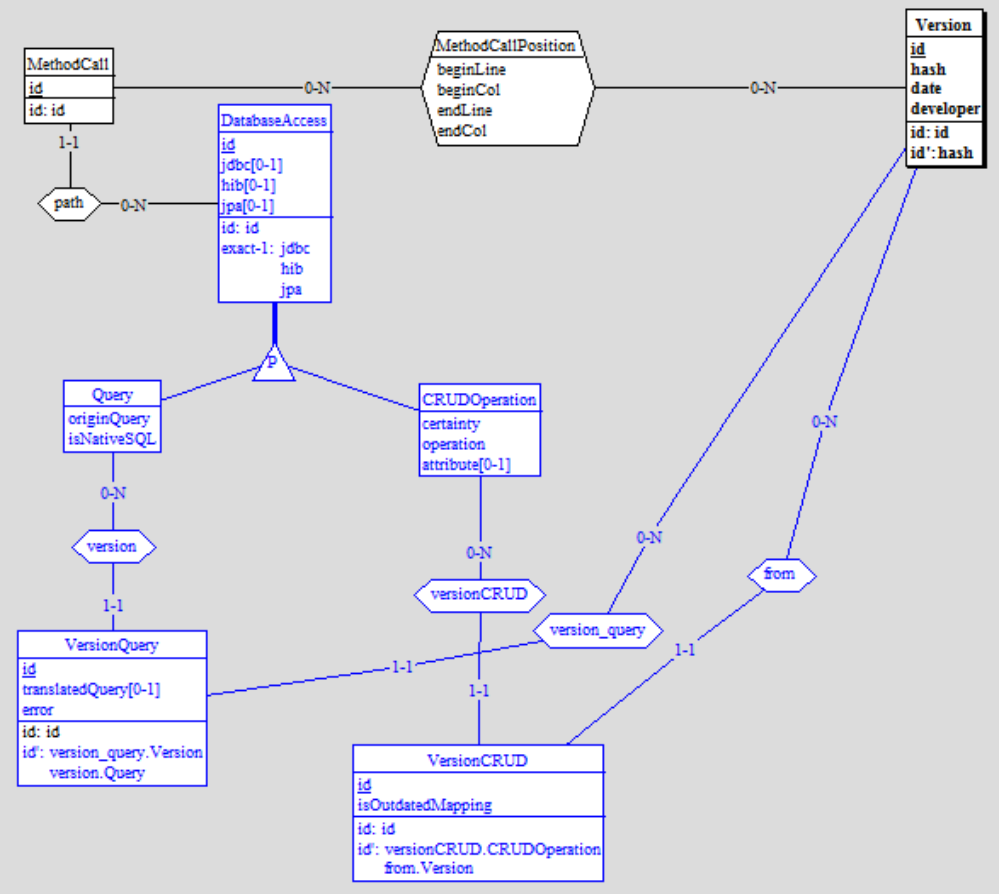

FIGURE 4.3 – Module d'historique des accès (Meurice et al.)

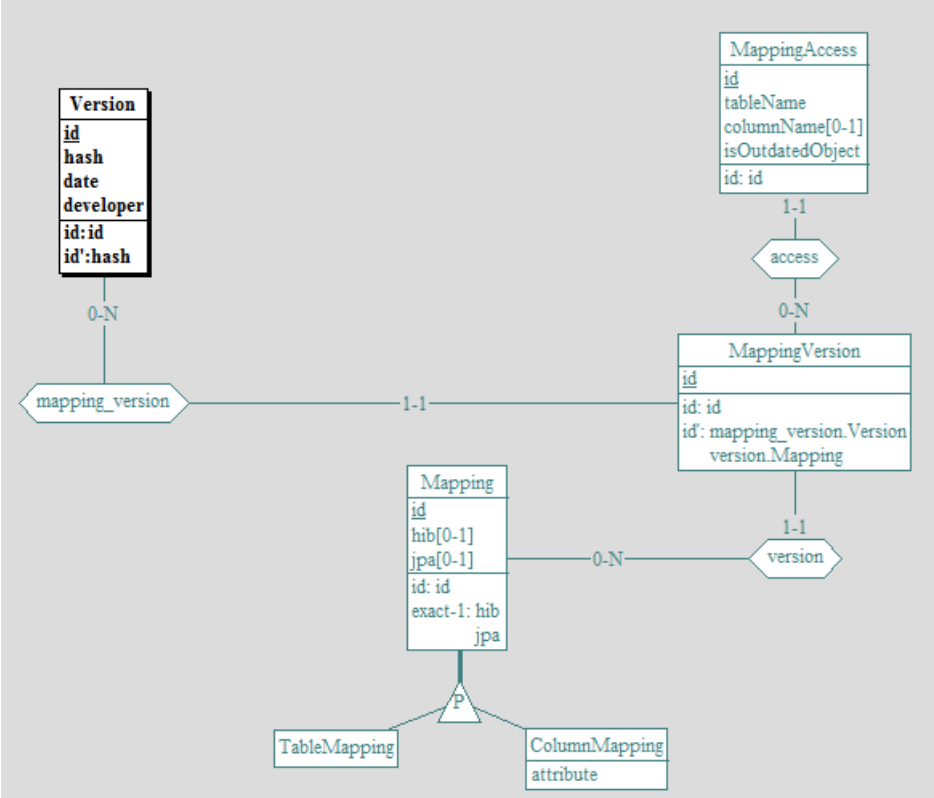

FIGURE 4.4 – Module d'historique des mappings (Meurice et al.)

Traduction des requêtes

Dans le méta-modèle présenté, l'analyse statique de Meurice et al. extrait les requêtes SQL mais également les requêtes écrites en JPQL et HQL, deux langages qui ont été présentés dans les concepts préliminaires de la Section 3.1. Dès lors, comme cela peut se remarquer sur la Figure 4.3, la table "*VersionQuery*" contient une colonne "*translatedQuery*" qui fait référence à la traduction de ces requêtes vers leur équivalence en SQL. Cette notion est importante, car la méthode proposée dans la suite du chapitre base son analyse uniquement sur les requêtes rédigées en SQL.

En ce qui concerne les requêtes écrites en HQL, leur traduction s'effectue en invoquant le compilateur d'HQL à SQL qui est interne à Hibernate. Ce dernier établit la correspondance selon le contexte d'exécution de la requête initiale. L'équivalence s'établit sur un principe similaire en ce qui concerne les requêtes en JPQL. Dès lors, ce sont ces résultats obtenus qui sont uti-

lisés en information d'entrée pour la méthode établie dans ce chapitre. Elle concentre alors son analyse sur ces requêtes SQL (sans considérer les opérations CRUD).

Finalement, cet aparté est important car il permet de souligner un point particulier : étant donné qu'il s'agit d'analyser des requêtes extraites sur base d'analyse statique et de traduction d'équivalence en SQL, les requêtes capturées ne sont pas forcément similaires à celles que le développeur consulte dans le code source. En effet, par exemple dans le cas des ORM, la requête obtenue et qui est analysée par la méthode est une requête SQL qui a non seulement été traduite mais également dont les mappings ont été déduits à l'exécution. Par conséquent, ce qui sera considéré par la suite comme des modifications de requêtes se réfère en réalité à des modifications dans les requêtes traduites vers le SQL : lorsqu'un mapping change, la requête traduite aura également changé, alors que du côté du programmeur il se peut qu'aucun changement n'ait été commis. Cela s'explique aisément par le fait qu'un mapping peut changer de manière assez invisible pour le programmeur dans le cadre des techniques d'ORM.

4.1.2 L'analyse d'origine, point de départ d'une réflexion

Tout d'abord, il est important d'insister sur le fait que l'analyse d'origine, telle que présentée dans la Section 3.2, requiert une approche *semi-automatique* afin d'être réellement efficace et utile. En effet, il n'existe pas de processus ou de critères strictement formels et fiables à 100% afin de certifier qu'une entité d'un programme est une version modifiée d'une ancienne entité. Il existe tout au plus des méthodes heuristiques qui permettent d'indiquer la probabilité d'une telle évolution. C'est pourquoi la motivation première, dans la recherche et l'objectif d'établir une méthode utile, est de proposer pour chaque requête d'accès un pourcentage de probabilité de son évolution.

Ensuite, le type d'heuristiques pour identifier l'évolution tel que proposée par Godfrey et al. [32], à savoir basées sur les appels entrants/sortants de fonctions, n'est pas applicable sur des requêtes d'accès à la base de données étant donné qu'il n'existe pas à proprement parler de notion de relation d'appels pour ces dernières. Tout au plus, il serait possible de considérer les

appels entrants/sortants des méthodes qui contiennent ces requêtes d'accès. Cependant, ces informations sont lourdes à calculer pour des systèmes de plusieurs dizaines de milliers de lignes de code. Par conséquent, l'objectif de la méthode est de pouvoir déterminer d'autres heuristiques sur base des informations minimales à disposition. A partir de cela, la méthode présentée dans ce chapitre a donc pour but d'adapter le champ d'application de l'analyse d'origine aux requêtes d'accès.

Finalement, et afin de fixer pour l'entièreté de la suite du raisonnement le terme d'évolution d'une entité (i.e. une requête d'accès dans le cas présent), la définition suivante est considérée :

Definition 4.1. *Supposons que R représente une requête d'accès à la base de données qui apparaît dans une version particulière V d'un logiciel. Si R n'apparaît pas identiquement dans une version V' antérieure ou postérieure, alors la requête R' est considérée comme une évolution de R lorsque R' est introduite dans V' en étant une version modifiée de R.*

4.1.3 Terminologie de l'évolution de requêtes

Au-delà de la définition 4.1 générale de l'évolution donnée pour une requête, il est nécessaire de la raffiner afin de définir également les différents types de sous-évolutions qui peuvent apparaître. Considérant deux versions différentes du logiciel *V* et *V'*, une évolution de requêtes s'envisage selon l'identification d'une des situations suivantes :

A. **Déplacement :**

Une requête *R1* est dite *"déplacée"* si elle est présente à la fois dans *V* et dans *V'*, et que son emplacement a été modifié. Le terme d'emplacement fait référence à la méthode dans laquelle la requête est contenue, ainsi que la ligne spécifique où elle est écrite et le nom du fichier contenant la méthode.

B. **Modification de requête :**

Une requête *R1* dans *V* est modifiée si une requête différente *R2* dans *V'* se situe au même emplacement que *R1* et produit une tâche similaire. La similarité de la tâche est évaluée selon le pourcentage de similarité des tables et colonnes sur lesquelles agit la requête.

C. **Modification de méthode :**

Une requête *R1* dans *V* a évolué si une requête *R2* dans *V'* (similaire ou non à R1) n'a pas été déplacée et produit une tâche similaire, moyennant la modification de la méthode qui contient la requête, qu'il s'agisse de ses paramètres, de son nom ou de son type de retour.

D. **Accès erronés :**

Une requête *R2* dans *V'* tente d'accéder en base de données à des tables ou des colonnes qui n'existent plus dans la version *V'*, tandis que *R1* pouvait y accéder avec succès dans *V*.

Les versions du système choisies peuvent être quelconques, et n'ont donc pas nécessairement besoin d'être deux versions successives. De plus, les situations définies ci-dessus sont également combinables : il est par exemple possible qu'une requête possède à la fois la caractéristique de *déplacement* et d'*accès erronés* lors de son évolution. Finalement, la méthode établie a évidemment pour objectif de reconnaître également la situation triviale (et non décrite ci-dessus) où une requête n'a pas été modifiée ou déplacée entre deux versions où elle est présente, ce qui représente le cas le plus classique à détecter, à savoir un maintien strict de la requête entre les deux versions. Bien qu'il pourrait être discutable de considérer ce genre de phénomène comme inutile lorsque l'on s'intéresse à l'évolution de requêtes, celui-ci a son importance : afin d'étudier le cycle de vie complet d'une requête, il est important de détecter lorsque celle-ci est présente sur deux versions successives sans subir de modifications particulières ni de déplacement, puisque ce cas de "maintien" de la requête est fort différent du cas où elle disparaît sur certaines versions et réapparaît ensuite (ce qui implique une suppression et un ajout à notifier).

4.2 Méthode d'analyse d'évolution

Cette section présente l'élaboration de la méthode d'analyse du cycle de vie des requêtes. Tout d'abord, une approche par heuristiques est proposée où les différentes heuristiques sont explicitées et illustrées par des exemples. Ensuite, cette approche est alors combinée avec la technique d'analyse et d'extraction de requêtes (et leur traduction en SQL au besoin) afin de proposer la méthode finale.

Comme mentionné précédemment, l'analyse d'origine doit être un processus semi-automatisé afin d'être efficace, étant donné que l'on ne peut pas identifier selon des critères entièrement formels l'évolution. Dès lors, une série d'heuristiques doivent être élaborées afin de se rapprocher avec la meilleure précision possible d'une identification formelle. Dans la suite, les heuristiques présentées sont systématiquement appliquées sur un couple de requêtes SQL issues de deux versions distinctes du logiciel considéré.

Durant tout le reste de la section, les deux requêtes de l'Exemple 4.1 permettront d'illustrer nos propos. Elles ont été créées via l'utilisation de JPA, avant d'être traduites dans leur équivalent en SQL lors de l'analyse statique. Nous nous référerons à ces requêtes par les abréviations R1 et R2.

Exemple 4.1.

R1 ≡ SELECT user_role_id, user FROM blc_user_role WHERE user = ? ;

R2 ≡ SELECT user_id, user_role_id FROM blc_user_role WHERE user_id = ? ;

4.2.1 Approche par heuristiques

Concept préliminaire

Il est tout d'abord nécessaire de présenter la notion de *longest common substring* (qui sera abrégée en LCS), car cette dernière constitue une notion de base utilisée à travers de nombreuses heuristiques. De manière concrète, le LCS est la plus longue chaîne de caractères qui est sous-chaîne commune à deux chaînes de départ. Afin d'illustrer ces propos, considérons le Tableau 4.1.

U	**S**	**E**	**R**	**_**	R	O	L	E	_	I	D
U	**S**	**E**	**R**	**_**	I	D					

Tableau 4.1 – Illustration du LCS

Les termes *USER_ROLE_ID* et *USER_ID*, deux noms de colonnes présents dans l'Exemple 4.1, sont nos chaînes de départ. Dans celles-ci, plusieurs sous-chaînes communes existent, comme par exemple *USE* ou *USER_*. Cette dernière étant la plus longue de toutes celles envisageables, il s'agit donc de notre *longest common substring*. Finalement, il faut également remarquer que cette sous-chaîne commune de caractères n'est pas forcément un préfixe ou un suffixe des termes de départ dans certains cas.

Dans le domaine d'analyse de dépôts liés au développement logiciel, il s'agit d'une mesure fréquemment utilisée, que nous avons décidé d'employer principalement pour deux raisons :

1. D'une part, il n'existe pas de méthode parfaite afin de déterminer si deux entités du logiciel sont identiques, dès lors la comparaison textuelle est une métrique souvent utilisée étant donné qu'elle est à la fois simple et plutôt fiable. Puisque ce genre d'analyse s'effectue sur des volumes de données d'une taille gigantesque, cette simplicité s'avère donc être un atout.

2. D'autre part, cette approche textuelle est adaptée au cas des requêtes d'accès à la base de données étant donné que les développeurs ont tendance à vouloir donner de la sémantique aux noms de colonnes et de

tables choisis. Par conséquent, comparer la similarité des noms apporte une estimation relativement bonne de la similarité des requêtes.

Heuristique 1 : Matching textuel de requêtes

La première heuristique applicable consiste à calculer la similarité textuelle entre deux requêtes via l'utilisation de la métrique du LCS. Puisque cette métrique prend en entrée deux chaînes de caractères, il faut pouvoir effectuer l'évaluation du LCS de manière isolée pour les tables et colonnes présentes dans les deux requêtes. Finalement, la longueur en taille des différents LCS obtenus est additionnée pour obtenir la valeur finale de similarité. De ce fait, cela permet de pouvoir comparer de manière plus précise les tables et colonnes accédées par les deux requêtes en questions.

Afin de mieux comprendre le fonctionnement de cette heuristique, reprenons les requêtes R1 et R2, et appliquons l'heuristique sur les colonnes présentes dans la clause *select* (le principe étant le même pour les autres clauses de la requête). La première étape consiste alors à créer des couples de chaînes de caractères, où le premier élément du couple appartient à R1 et le second élément appartient à R2. Le Tableau 4.2 reprend les différentes combinaisons possibles de couples, où chaque couple est identifié soit par une astérisque (*) pour le premier couple, soit par deux astérisques (**) pour le second couple.

	user_role_id (R1)	user (R1)	user_id (R2)	user_role_id (R2)	TOTAL
COMB1	*	**	*	**	5 + 4 = 9
COMB2	*	**	**	*	12 + 4 = 16

Tableau 4.2 – Combinaisons possibles - Exemple

Une fois les couples déterminés, le total est calculé de la manière suivante : on détermine le LCS de chaque couple ainsi que sa longueur en nombre de caractères, avant d'additionner finalement les différentes longueurs obtenues. Par exemple, dans le cas de la première combinaison, le LCS pour le premier couple ("user_role_id","user_id") donne la chaîne "user_" de longueur 5, et le LCS du second couple ("user","user_role_id"") donne la chaîne "user" de

longueur 4, rapportant le total final à 9.

Plusieurs observations s'imposent alors à présent sur cette heuristique. Le choix de travailler par combinaisons possibles de couples est important, sans quoi une ambiguïté apparaîtrait. En effet, le problème est le suivant : les colonnes ou tables dont les deux requêtes font l'objet de la comparaison se présentent sous un ordre totalement aléatoire. Or, l'objectif étant d'obtenir le score total le plus élevé possible, il faut pouvoir s'assurer de comparer deux à deux les colonnes ou tables qui donnent le meilleur score final possible, comme le démontre le Tableau 4.2. En effet, on remarque que la valeur totale obtenue est différente selon les combinaisons choisies, mettant en lumière la problématique exposée précédemment. Dès lors, il est donc important de pouvoir considérer les différentes combinaisons possibles afin de garder la valeur totale optimale. Le fait de considérer toutes les combinaisons possibles pour obtenir la valeur maximale est purement théorique dans le cadre de l'heuristique, et le choix de l'implémentation peut donc varier pour obtenir la valeur optimale. Sans rentrer dans les détails, pour un nombre important de combinaisons, le développeur pourrait choisir d'implémenter son algorithme avec une stratégie permettant de réduire le nombre de calculs effectués dans la recherche de la valeur optimale, afin de réduire la complexité de l'algorithme. Cette question purement pratique est donc laissée à l'appréciation du développeur et n'est pas discutée plus amplement.

Finalement, une fois le meilleur total obtenu, il est pondéré par la taille entière (en nombre total de caractères) des requêtes de la manière suivante :

$$RequestMatching \equiv \frac{total \times 2}{longueur(R1) + longueur(R2)} \tag{4.1}$$

où la fonction *longueur(X)* renvoie, pour la chaîne X, le nombre total de caractères qu'elle contient.

Cette formule est une moyenne pondérée, et son utilisation se justifie par le fait que la comparaison des tables et colonnes accédées par les requêtes est un premier bon indice de l'évolution d'une seule et même requête. En

effet, deux requêtes peuvent être liées si cette métrique est proche de 1 étant donné qu'elles accèdent donc à des colonnes et tables similaires, impliquant le fait qu'elles ont dès lors des tâches potentiellement similaires. La similarité d'une tâche s'évalue donc en fonction de la similarité des éléments auxquels on accède, et ainsi qu'en fonction des conditions d'accès dans la requête (en ce qui concerne la clause *where* par exemple).

Finalement, la problématique à régler afin que cette heuristique donne des résultats optimaux est de déterminer le seuil minimum à partir duquel la métrique est fiable. En effet, si le seuil minimum fixé pour déterminer un matching positif est très proche de 1, cela signifie que l'ensemble des matchings de requêtes qui sont conservés est potentiellement inclus dans l'ensemble des matchings de requêtes qui doivent effectivement être conservés dans les faits réels, ayant donc pour effet de ne pas prendre en compte la totalité des requêtes susceptibles d'avoir effectivement évolué. A contrario, fixer le seuil minimum à une valeur trop basse impliquerait la prise en compte de matchings inutiles vis-à-vis de la réalité.

Dès lors, au travers de recherches empiriques et d'application sur l'étude de cas présentée au Chapitre 5, les divers résultats obtenus tendent à montrer qu'un seuil minimum fixé à 0.5 permet à la fois de conserver tous les matchings intéressants et de limiter la prise en compte de matchings inutiles, bien que cette problématique sera discutée lors de l'évaluation de la méthode.

Heuristique 2 : Matching des méthodes

Considérons m_1 et m_2, deux méthodes qui contiennent respectivement les requêtes *R1* et *R2*. Le matching de méthodes est effectué en utilisant le principe identique à celui de l'heuristique précédente.

Concrètement, là où le LCS était évalué auparavant sur les tables et colonnes des requêtes, il est à présent calculé séparément pour le nom de la méthode, les paramètres de la méthode (ainsi que leurs types) et le type de retour, afin de déterminer le taux de similarité des deux méthodes m_1 et m_2. Pour reprendre l'exemple de départ, voici les méthodes qui contenaient R1 et R2 :

Exemple 4.2.

M1 : *public List<UserRole> readUserRolesByUserId(final Long userId)*

M2 : *public List<UserRole> readUserRolesByUserId(Long userId)*

Appliquer cette deuxième heuristique sur l'Exemple 4.2 revient donc à évaluer le LCS pour le couple de noms de méthode de M1 et M2, le couple de types de retour, et le couple de paramètres (ou les combinaisons possibles de couples lorsqu'il y a plus d'un paramètre). La suite du déroulement est similaire à celui de l'heuristique 1 : les longueurs des LCS obtenus sont additionnées pour obtenir le total. Finalement, ce total est à nouveau pondéré de manière identique, cette fois-ci en fonction des méthodes et non des requêtes :

$$MethodMatching \equiv \frac{total \times 2}{longueur(M1) + longueur(M2)} \tag{4.2}$$

De plus, dans le cas où le *Method Matching* donne un score inférieur à 100%, il est intéressant de conserver l'information sur la manière dont la méthode a changé, à savoir si il s'agit par exemple uniquement de changements dans les paramètres (tel qu'un ajout) ou si le nom de la méthode a également changé. Cette information peut présenter une certaine importance étant donné qu'en général, un simple ajout de paramètre présente potentiellement moins d'impact sur l'évolution de la requête par rapport à un nom de méthode modifié qui implique bien souvent des changements sur les traitements qu'elle a pour but d'effectuer, et donc également sur le traitement effectué par la requête contenue.

Heuristique 3 : Matching des chemins d'accès

Cette heuristique emploie exactement le même principe qu'auparavant, cette fois-ci concernant les chemins d'accès respectifs des fichiers contenant les requêtes. Cela permet de déterminer si la requête a été déplacée d'un fichier à un autre entre deux versions du système. Dans ce cas-ci, les LCS sont évalués en comparant chaque couple de répertoire et de fichier présents dans le chemin d'accès, et sont finalement additionnés pour obtenir le total. La formule finale est donc la suivante, en considérant les chemins d'accès A1

et A2 respectivement relatifs à R1 et R2 :

$$FilePathMatching \equiv \frac{total \times 2}{longueur(A1) + longueur(A2)} \tag{4.3}$$

Au final, cela permet d'obtenir un taux de similarité élevé dans le cas où un des deux chemins d'accès comparés aurait simplement rajouté un répertoire supplémentaire entre les deux versions, tout en indiquant cet ajout.

4.2.2 Élaboration de la méthode

L'articulation des heuristiques présentées au préalable permet dès lors la création d'une méthode d'analyse d'évolution des requêtes. Cette articulation, bien qu'elle permet de s'orienter vers les résultats escomptés, est également accompagnée d'autres informations qui viennent renforcer l'accomplissement de l'objectif initial grâce au méta-modèle présenté dans la Section 4.1.1.

Tout d'abord, la méthode récolte en information d'entrée dans le méta-modèle les requêtes extraites du code source du système concerné (via l'analyse statique). Ensuite, une fois ce traitement préalable effectué, elle a pour objectif d'établir la liste des requêtes en évolution entre deux versions du logiciel. Elle prend donc en entrée la liste totale des requêtes traduites en SQL pour chacune des versions, ainsi que les informations qui y sont reliées et présentes dans le méta-modèle. Dans ces informations, on retrouve par exemple leur emplacement dans le code, la méthode qui les contient, etc. Il est donc important de remarquer que deux requêtes textuellement identiques sont considérées comme différentes durant l'analyse, et ce à partir du moment où elles sont par exemple déclenchées à deux endroits différents du code. En effet, les distinguer possède son importance : leur cycle de vie dans le logiciel, qui est la donnée à analyser qui nous intéresse, n'est pas forcément identique.

A partir de cela se dégage une première ossature principale à établir pour la méthode. Ces listes de requêtes sont soumises successivement aux trois heuristiques présentées dans la Section 4.2.1, produisant au final des résultats par couple de requêtes (R1,R2), où la première appartient à l'une

des versions et la deuxième à l'autre version. Bien que les détails purement théoriques sont ceux qui nous intéressent, il faut remarquer du point de vue pratique que calculer exhaustivement ces résultats peut s'avérer gourmand en temps pour un nombre élevé de couples de requêtes. Dès lors, l'aparté sur cette problématique reste en suspens, et sera discuté dans la Section 4.3 consacrée à l'implémentation.

Ensuite, les résultats obtenus par les heuristiques donnent alors trois premiers "scores" qui servent d'indication sur trois éléments :

A. La similarité des colonnes et tables accédées par les requêtes, selon l'*heuristique 1*.

B. La similarité des tâches à effectuer par les méthodes qui contiennent ces requêtes, selon l'*heuristique 2*.

C. La proximité de leur emplacement, selon l'*heuristique 3*.

A partir de ces informations récoltées, où chaque score représente un taux de similarité compris entre 0 et 1, la méthode consiste alors à remettre sur une moyenne arithmétique ces scores afin d'avoir la probabilité d'évolution finale des requêtes. La possibilité de pondérer cette moyenne a été envisagée avant d'être écartée. En effet, il pourrait sembler dans un premier temps qu'accorder plus d'importance dans la pondération aux tables et aux colonnes accédées par les requêtes donnent une probabilité finale plus précise. Cependant, les résultats empiriques obtenus sur les systèmes de départ indiquent que cette pondération particulière n'est pas nécessaire : l'emplacement de la requête a autant d'importance que les tables et colonnes accédées, sans quoi deux requêtes présentes à quelques lignes d'intervalle de code déclencheraient des erreurs dans l'analyse.

Ensuite, une fois ce score final obtenu pour tous les couples possibles de requêtes, l'étape suivante de la méthode consiste à ne conserver que les "meilleurs" couples selon les critères suivants :

Critère 4.1. *Un couple final de requêtes doit contenir obligatoirement une requête de la version V et l'autre de la version V'.*

Critère 4.2. *Chaque requête de la version V ne peut être contenue que dans un seul couple final, ce dernier étant celui qui possède le score le plus haut.*

Critère 4.3. *Lorsqu'une requête n'est associée à aucun couple final, elle est étiquetée en tant que requête nouvellement ajoutée ou supprimée.*

Finalement, cette avant-dernière étape permet donc de récolter le traçage de l'évolution des requêtes entre les deux versions. A partir de cela, l'ultime étape consiste à itérer la méthode depuis le début sur le nombre de versions souhaité. De cette manière, l'objectif initial est alors accompli : le cycle de vie d'une requête peut être retracé sur plusieurs versions. Le récapitulatif du déroulement de la méthode est illustré sur la Figure 4.5.

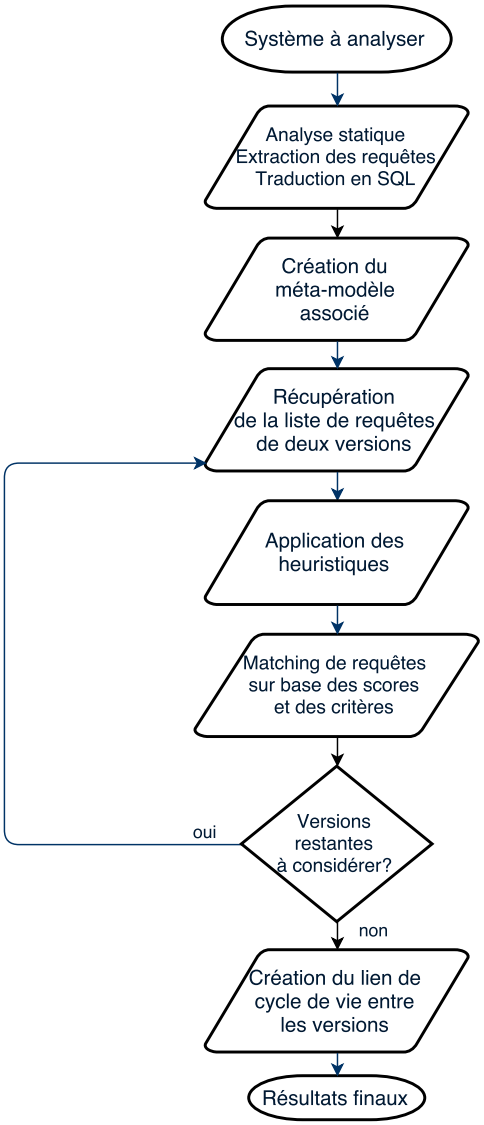

FIGURE 4.5 – Illustration de la méthode d'analyse du cycle de vie des requêtes

Remarques finales

Lors de la phase de la création d'une approche par heuristiques, notre réflexion s'est également portée sur d'autres heuristiques intéressantes qui n'ont pas été retenues pour diverses raisons (tel que le fait de ne pas disposer aisément des informations nécessaires pour les mettre en place). Cependant, il semble intéressant de les mentionner brièvement afin d'en faire éventuellement une utilisation future. Parmi celles-ci, citons notamment deux d'entre

elles.

La première heuristique intéressante afin de renforcer l'analyse de l'évolution des requêtes consiste à comparer en parallèle l'évolution du schéma de la base de données. A travers les modifications apportées au schéma, des corrélations peuvent être établies au regard des modifications des requêtes dans le code. En effet, l'ajout, la suppression ou le renommage d'un élément de la base de données a inévitablement des répercussions sur les requêtes qui accédaient à ces éléments.

La seconde heuristique consiste à récupérer l'ensemble des méthodes appelantes et appelées de la méthode qui contient la requête qui fait l'objet de l'analyse. Grâce à cet ensemble, il est possible de le comparer à celui de la seconde requête afin d'avoir un indicateur de plus sur la probabilité qu'il s'agit d'une évolution de requête. En effet, si ces ensembles contiennent un nombre élevé d'appelants et d'appelés similaires, cela augmente les chances qu'il s'agisse de la même méthode dans les deux cas, et donc également que la requête contenue soit identique.

4.3 Implémentation de la méthode

La méthode d'analyse du cycle d'évolution des requêtes, dont les concepts théoriques sont présentés dans la Section 4.2, ont fait l'objet d'une implémentation de notre part. Cette implémentation a pour objectif de pouvoir généraliser l'analyse de l'évolution du cycle des accès pour tout système, à condition que les données en entrée soient conformes au méta-modèle de la Section 4.1.1.

Les choix des technologies utilisées et d'une architecture justifiée sont présentés dans la Section 4.3.1, tandis que certaines considérations algorithmiques sont décrites dans la Section 4.3.2. Finalement, les fonctionnalités de la méthode outillée implémentée sont illustrées dans la Section 4.3.3.

4.3.1 Choix technologiques et architecturaux

La méthode a été implémentée sur base du choix d'une architecture *3-tier*. Il s'agit d'une architecture qui permet de séparer en 3 couches distinctes les responsabilités à endosser. Ces couches sont les suivantes :

A. La couche présentation : elle se charge de présenter l'information au client tout en permettant l'interaction avec ce dernier. Les requêtes du client sont donc renvoyées vers la couche métier avec laquelle elle communique, cette dernière lui renvoyant les informations qu'elle a traitées au préalable.

B. La couche métier : le serveur possède la responsabilité d'effectuer le traitement applicatif, à savoir gérer la logique applicative centrale. Cette couche répond aux requêtes du client, en traitant les données issues de la couche d'accès aux données en fonction des calculs et tâches à effectuer sur ces données. Elle sert donc de relais et d'orchestration entre les deux autres couches.

C. La couche d'accès aux données : le serveur de base de données possède les données persistantes utiles pour la couche métier.

L'avantage du choix d'une telle architecture est qu'elle permet d'organiser l'approche en plusieurs couches dont les responsabilités et services proposés

sont délimités. De plus, ces couches sont établies de manière hiérarchique, de telle sorte qu'une couche inférieure propose ses services aux couches supérieures. En effet, dans notre cas, la couche d'accès aux données met à disposition les données persistantes à la couche métier, qui traite ensuite ces dernières pour permettre leur présentation au client. Dès lors, cette découpe des responsabilités permet de développer et de faire évoluer chaque couche indépendamment sans répercussion, et donc avec un couplage faible. Cela permet finalement d'obtenir un modèle souple et une maintenance facilitée.

Au niveau de la couche d'accès aux données, le patron de conception implémenté est le *Data Access Object*, qui permet de se rendre indépendant de la manière dont sont stockées les données au niveau des objets métier. Cela implique de regrouper la gestion de l'accès aux données à un seul et même endroit, plus facile à faire évoluer que lorsque ces accès sont disséminés dans le code métier.

Finalement, sur base de cette architecture, les choix technologiques suivant ont été établis :

— Au niveau de la couche métier, le coeur de l'application Web est écrit en Java (JDK1.7) et tourne sur un serveur Apache Tomcat (v7.0). De plus, afin de pouvoir décomposer les requêtes SQL pour les analyser, le coeur métier utilise la librairie JSqlParser afin de parser le contenu syntaxique des requêtes. Le patron de conception Visiteur a été mis en place afin de construire notre parser final. Finalement, la libraire JGit a également été utilisée afin de récupérer les commits effectués par les développeurs sur les systèmes open-source analysés et extraire les changements entre commits (à savoir la fonction *diff* de Git).

— Concernant la couche présentation, l'application propose une interface pour l'utilisateur sous forme de HTML, CSS et d'utilisation de JavaScript. Des JavaServer Pages (i.e. JSP) sont utilisées pour rajouter du contenu dynamiquement au contenu statique de base des pages.

— Finalement, la couche d'accès aux données repose sur un serveur MySQL Community (5.6). La base de données MySQL contient le schéma re-

latif au méta-modèle introduit précédemment. La libraire JDBC a été utilisée au niveau de la couche supérieure afin de communiquer avec la base de données.

4.3.2 Choix algorithmiques

Bien qu'il n'est pas nécessaire de s'étendre en long et en large sur la totalité des choix d'implémentation réalisés, quelques précisions s'imposent à certains niveaux. Ainsi, il semble important de mentionner une problématique déjà évoquée brièvement dans la Section 4.2.1 : lorsqu'il y a beaucoup de requêtes à analyser et que celles-ci contiennent un grand nombre de tables et de colonnes accédées, cela donne rapidement un nombre énorme de combinaisons de couples de colonnes/tables à comparer lors de la recherche du *longest common substring* optimal. Les systèmes à analyser peuvent comporter des historiques très lourds lorsqu'ils contiennent des millions de ligne de code. Ces problèmes de volumétrie sont donc abordés dans cette section.

Dans la méthode théorique présentée auparavant, l'objectif est de calculer les formules des heuristiques pour chaque couple de requêtes de manière exhaustive. Cependant, afin de réduire les temps de calcul dans la pratique, il est possible d'éliminer au fur et à mesure de l'algorithme certains couples dont on peut présumer avec assez de certitude qu'ils ne mèneront pas au final à détecter une évolution de requêtes. Pour y parvenir, la stratégie mise en place est la suivante : au départ, la première heuristique est calculée pour la totalité des combinaisons de requêtes car il est impossible de pouvoir prédire directement quelles requêtes sont enclines à avoir évolué. Ensuite cependant, ce premier score récolté sert d'indice pour élaguer la suite des calculs à effectuer. L'élagage s'établit donc en fonction de la valeur de la première heuristique : toutes les combinaisons donnant un score inférieur à une certaine valeur implique leur élimination des futurs calculs. L'Algorithme 1 présente cette stratégie telle qu'elle fut implémentée.

Algorithm 1 Stratégie d'analyse d'évolution des requêtes

Require: $L1 \neq \varnothing \wedge L2 \neq \varnothing$
{L1, L2 respectivement l'ensemble de requêtes des versions}

init $L3 = \varnothing$ {ensemble de triplets (rq1 x rq2 x score-heuristique)}
init threshold {seuil d'acceptation de l'heuristique 1}

for all ($rq1 \in L1$, $rq2 \in L2$) **do**
  $score \leftarrow heuristique1(rq1, rq2)$
  $(rq1, rq2, score) \in L3$
end for

for all $(rq1, rq2, score) \in L3$ **do**
  if $score < threshold$ **then**
    $L3 \leftarrow L3 \setminus (rq1, rq2, score)$ {élagage}
  else
    ... {le calcul des heuristiques suivantes est poursuivi}
  end if
end for

Finalement, concernant l'algorithme 1, l'élagage s'établit sur base de la valeur *threshold*. Dès lors, la problématique de l'élagage devient la problématique de déterminer le seuil minimal de confiance pour conserver le couple de requêtes. Par plusieurs expériences et tests empiriques, dont les détails sont disponibles dans le Chapitre 5, un seuil fixé à 0.50 permet de conserver tout candidat potentiel d'évolution.

4.3.3 Méthode outillée et fonctionnalités

Au terme de l'implémentation, une méthode outillée complète et fonctionnelle a été créée contenant à la fois la méthodologie proposée, mais également quelques autres fonctionnalités additionnelles. Elle est également disponible en CD annexe au mémoire.

Cette méthode outillée est déjà opérationnelle pour les bases de données d'historique de deux systèmes présentés lors de l'évaluation. Elle permet d'analyser l'évolution de requêtes entre deux versions au choix du système, en filtrant également la recherche via le seuil minimal requis de similarité (cf. Figure 4.6). De plus, afin d'affiner ces résultats, d'autres informations

sont également prises en compte telles que la détection des tentatives d'accès à des éléments de la base de données qui n'existent pas (ou plus) pour trouver une piste d'*erreur d'accès* d'une requête. La méthode outillée détecte également les copier-collers de requêtes faits par les programmeurs, et leurs duplications aux divers endroits du code, tout en proposant de les visualiser directement sur les répertoires GitHub correspondants. La migration d'une requête d'une technologie vers une autre est également prise en compte et affichée à l'utilisateur au besoin.

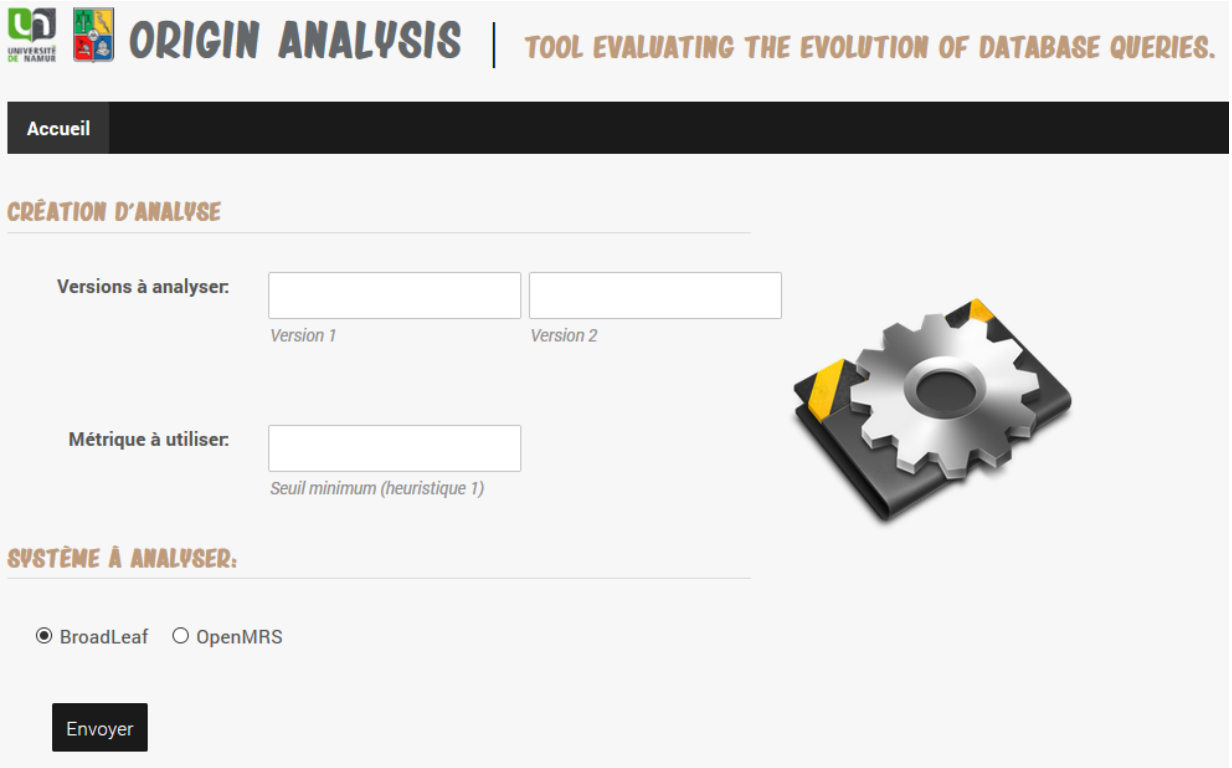

FIGURE 4.6 – Illustrations de l'outil - Page d'accueil

Ensuite, les résultats obtenus par la méthode sont divisés en plusieurs fonctionnalités. La Figure 4.7 présente un aperçu général qui groupe divers pourcentages. Les requêtes rejetées lors de l'analyse réfèrent à des requêtes syntaxiquement incorrectes en SQL, un cas qui peut arriver lorsque celles-ci sont écrites entièrement à la main par le développeur. Les requêtes défectueuses font quant à elles référence aux requêtes qui ont été traduites en SQL depuis une technologie comme Hibernate par exemple, et dont le résul-

tat de cette traduction donne une requête incorrecte syntaxiquement. Ce cas peut arriver lorsqu'il y a par exemple au départ des mappings défectueux au niveau de l'utilisation de l'ORM.

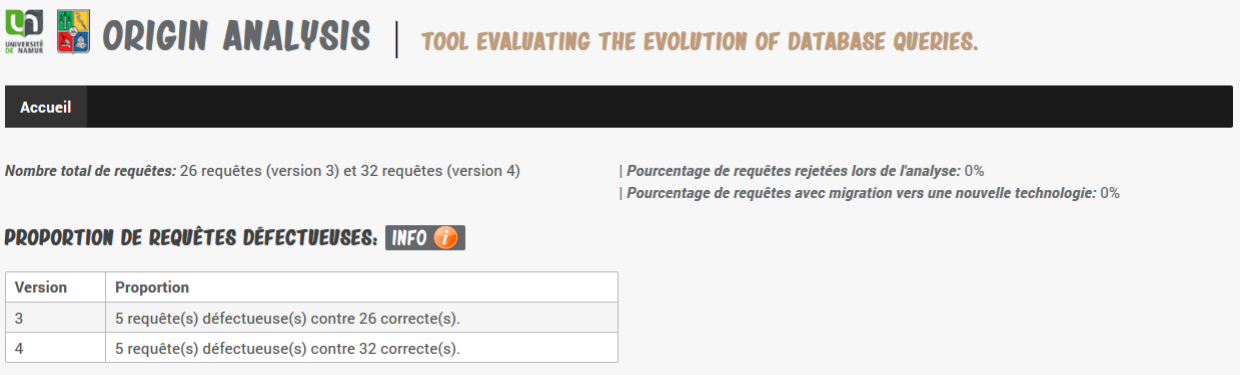

FIGURE 4.7 – Illustrations de l'outil - Résultats obtenus (1 sur 4)

La Figure 4.8 présente la répartition de l'évolution des requêtes, où 3 scenarii sont possibles : soit la requête est restée totalement identique entre les deux versions (cas d'évolution triviale), soit elle a évolué (i.e. cas de matching inférieur à 100%), et le pourcentage restant se réfère aux requêtes qui ont été supprimées ou ajoutées entre les deux versions. On retrouve également la répartition du matching des requêtes face à l'heuristique 1, dont l'information n'est pas utile pour le développeur mais permet de constater que les choix d'implémentation exposés précédemment permettent de réduire les combinaisons de requêtes à analyser. Finalement, lorsque des requêtes ont matché, un détail est proposé sur le type d'évolution subi : il peut s'agir d'un simple déplacement, d'un ajout de colonne/table, etc. Dans le cas d'une modification de la méthode qui la contient, d'autres informations sur la méthode sont également proposées. Ce type d'informations peut par exemple permettre d'analyser si les requêtes qui ont matché l'ont fait par des déplacements de portions de code, ou bien plutôt par des modifications en elles-même des tables/colonnes accédées qui impliquent souvent plus de conséquence au niveau des mappings par exemple.

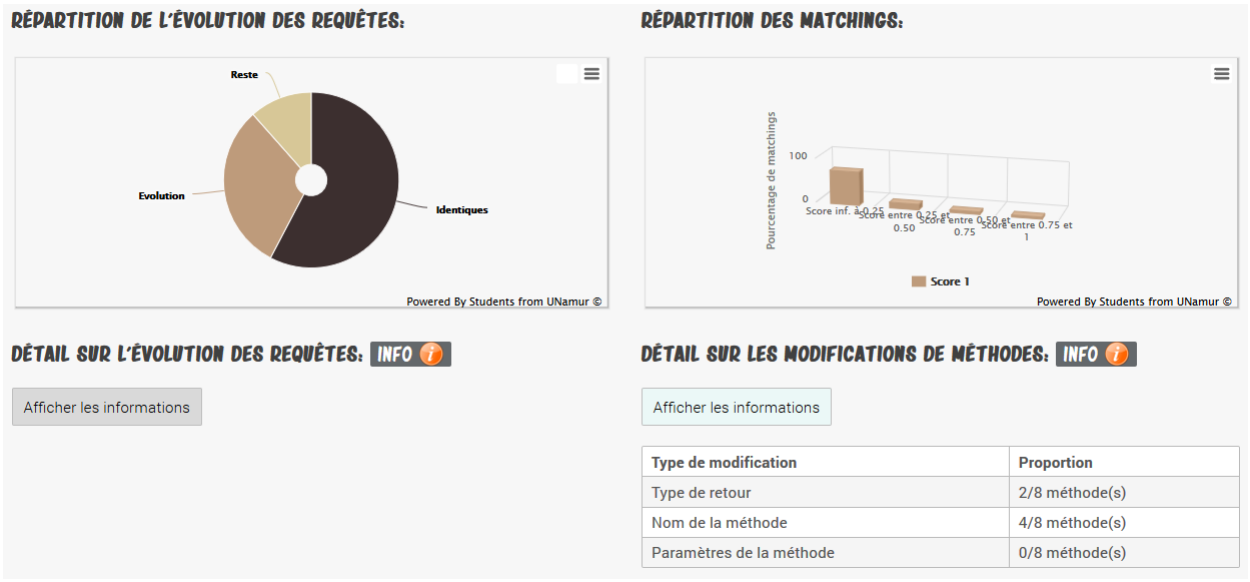

FIGURE 4.8 – Illustrations de l'outil - Résultats obtenus (2 sur 4)

De plus, la Figure 4.9 affiche finalement les matchings finaux de requêtes entre les deux versions analysées. La probabilité d'évolution indique le score final qui permet de mettre la puce à l'oreille du développeur quant à la similarité entre les deux requêtes. Dans le cas présent, on remarque que la requête de la version 4 est effectivement une version mise à jour de la requête de la version 3, où la colonne *phone_id* a été rajoutée par le développeur.

FIGURE 4.9 – Illustrations de l'outil - Résultats obtenus (3 sur 4)

Finalement, la Figure 4.10 propose de consulter la liste des requêtes qui ont été copié-collées entre les deux versions choisies. Dès lors, on remarque ici que la requête, qui a pour objectif de récupérer tous les *sku* présents, a été copié-collée dans une autre méthode. En consultant le lien qui rapporte au code complet de la classe sur GitHub, on remarque finalement que les deux méthodes créés par le développeur sont en fait identiques, au point près qu'elles envoient en valeur de retour soit un *"query.getSingleResult();"*, soit un *"query.getResultList();"* selon le nombre de *skus* souhaités.

DÉTECTION DE REQUÊTES COPIÉES-COLLÉES:

Requête de la version 3	**Requêtes de la version 4**
SELECT sku FROM org.broadleafcommerce.catalog.domain.SkuImpl sku (voir sur Github) **Chemin:** /BroadleafCommerce/src-framework /org/broadleafcommerce/catalog /dao/SkuDaoJpa.java **Méthode:** public Sku readFirstSku() **Technologie:** JPA	SELECT sku FROM org.broadleafcommerce.catalog.domain.SkuImpl sku (voir sur Github) **Chemin:** /BroadleafCommerce/src-framework /org/broadleafcommerce/catalog /dao/SkuDaoJpa.java **Méthode:** public List<Sku> readAllSkus() **Technologie:** JPA

FIGURE 4.10 – Illustrations de l'outil - Résultats obtenus (4 sur 4)

Chapitre 5

Évaluation de la méthode

La méthode élaborée et présentée dans le Chapitre 4 a fait ensuite l'objet d'une étude plus pratique. Cette dernière, réalisée à partir de données récoltées sur des systèmes utilisés réellement dans la vie courante, est dès lors détaillée à présent.

Tout d'abord, le cas d'étude choisi est introduit dans la Section 5.1. Cette section comporte la présentation des systèmes analysés, ainsi qu'une remise en contexte de leur utilisation et de l'approche abordée pour les analyser. Ensuite, avant de passer à l'évaluation du cas d'étude, la Section 5.2 discute la problématique du choix d'un seuil pour le calcul des heuristiques. Une fois ce seuil fixé, il est désormais possible de produire l'évaluation. Celle-ci, alors présentée dans la Section 5.3, se décompose de la manière suivante : la méthode d'évaluation et ses objectifs sont d'abord explicités, avant d'en présenter les résultats qui en découlent. Un regard critique est alors également porté sur l'évaluation, la manière dont elle se dirige et ses résultats. Finalement, un recensement de plusieurs cas de faux positifs type est proposé, afin de les documenter.

5.1 Étude de cas

5.1.1 Présentation des systèmes analysés

Une base de données a été générée pour deux systèmes sur base du méta-modèle énoncé dans la Section 4.1.1 : *OpenMRS* et *Broadleaf Commerce*. Elles contiennent chacune les différentes données historiques respectives de ces systèmes. Il est donc intéressant de présenter ces différents systèmes, tous deux étant utilisés en situation réelle, avant de passer à leur analyse.

Tout d'abord, *OpenMRS* est une plateforme open-source relative à la gestion de dossiers médicaux électroniques, développée depuis 2006. Le but recherché par cette plateforme collaboratrice est d'améliorer la qualité des services de santé offerts dans les pays en voie de développement. Cet objectif est atteint en créant un service aux coûts réduits et qui diminue les disparités entre environnements riches en ressources et environnements plus démunis.

Le système d'OpenMRS repose sur une base de données conceptuelle conçue pour permettre de ne pas être dépendant du type d'information médicale qui doit être collecté. Cela autorise la plateforme à être customisable pour différents usages. Pour parvenir à cette fin, la plateforme utilise le principe d'un dictionnaire de concepts : il s'agit d'un dictionnaire central où sont définies au préalable toutes les données destinées à être stockées. Au final, la plateforme permet donc de conserver tout type de diagnostique, procédure ou médicament. Cette pratique a deux avantages : elle permet d'éviter de modifier la structure de la base de données pour ajouter de nouveaux types de diagnostics par exemple, et elle permet également de partager facilement des dictionnaires de données entre tiers. Finalement, remarquons qu'OpenMRS utilise une base de données MySQL, principalement accédée via l'utilisation d'Hibernate et de SQL dynamique.

Ensuite, le second système faisant l'objet de l'étude est *Broadleaf Commerce*, un framework open-source développé en Java depuis 2008 et qui concerne le e-commerce. Le système a été conçu pour faciliter le développement de sites orientés commerce "business-to-business" et "business-to-customer", en proposant un modèle de données robuste et les services traditionnels associés à la vente. En bref, il s'agit d'une plateforme qui réduit la complexité du com-

merce multi-canaux tout en étant tout à fait customisable et extensible pour ses utilisateurs. Ce dernier a été développé en utilisant une base de données relationnelle dont les accès sont créés grâce à JPA.

Au niveau de la comparaison des deux systèmes, le Tableau 5.1 permet de constater qu'il s'agit de projets d'ampleur moyenne comptant une centaine de tables et quelques centaines de milliers de lignes de code. Ils sont développés depuis suffisamment de temps pour faire l'objet d'une analyse relativement solide.

Système	**Lignes de code**	**Tables**	**Colonnes**
OpenMRS	301 232	88	951
BroadLeaf	254 027	179	965

Tableau 5.1 – Systèmes analysés - Métriques de taille (Meurice et al.) [28]

Les différentes manières d'accéder à la base de données ont été mentionnées au préalable, et le Tableau 5.2 fait apparaître la variété des technologies utilisées entre les systèmes. D'une part, OpenMRS contient des accès Hibernate principalement, ainsi que quelques requêtes via JDBC, tandis que BroadLeaf contient uniquement des accès en JPA. Cela permet d'avoir une palette contenant les trois technologies.

Système	**Accès BD**		
	JDBC	**Hibernate**	**JPA**
OpenMRS	77	687	0
BroadLeaf	0	0	930

Tableau 5.2 – Systèmes analysés - Nombre d'emplacements d'accès à la base de données par technologie (Meurice et al.) [28]

Finalement, au niveau du nombre de tables et de colonnes de la base de données accédées par chacune des technologies pour OpenMRS, la répartition est la suivante : 2 tables et 10 colonnes sont accédées par JDBC, 14 tables et 46 colonnes par les deux technologies, tandis que le reste concerne uniquement Hibernate qui représentent donc le coeur.

5.1.2 Mise en contexte du cas concret

Le cas pratique se centre sur les deux systèmes présentés en détail dans la Section 5.1.1, à savoir BroadLeaf Commerce et OpenMRS. L'analyse statique des requêtes a donc été établie et est présente dans le méta-modèle de la Section 4.1.1. Initialement, BroadLeaf Commerce contenait 118 versions réparties équitablement dans le temps durant la période de février 2009 à mars 2015, tandis qu'OpenMRS en contenait 164 de décembre 2007 à octobre 2015. Cependant, le nombre de versions a ensuite dû être réduit sensiblement pour les raisons qui suivent :

Lors des premières phases d'analyse des résultats portant sur ce cas d'étude, la méthode fit apparaître que certaines requêtes des deux systèmes apparaissaient et disparaissaient de manière répétitive une version sur deux. Ce phénomène particulier attira donc finalement l'attention sur une anomalie détectée sur ces données de départ. La répétition d'apparitions et de disparitions était due au fait que les données initiales ont été récupérées sur deux branches Git distinctes pour certaines versions. Dès lors, l'analyse effectuée par la méthode outillée a permis d'exclure ces versions du cas d'étude final. Par conséquent, celui-ci repose alors sur 84 versions disponibles pour BroadLeaf Commerce, et 138 pour OpenMRS.

5.2 Problématique du choix d'un seuil

La méthode, appliquée au cas d'étude, permet de dévoiler entre autres le cycle d'évolution des requêtes sur les versions historiques successives. Parmi les résultats obtenus, une problématique restée ouverte dans le Chapitre 4 peut être désormais discutée sur base des phénomènes empiriques observés. Cette problématique réfère au choix du seuil minimum à considérer lors du calcul de l'heuristique 1 (cfr. Section 4.2.1), afin d'établir à partir de quelle valeur obtenue deux requêtes sont effectivement susceptibles d'être identiques.

Pour répondre à cette question, le pourcentage de requêtes qui ont matché entre deux versions successives a été calculé pour différents seuils minimums. Ces différents seuils ont fait apparaître les résultats synthétisés des Tableaux 5.3 (BroadLeaf) et 5.4 (OpenMRS). La version n dans la colonne de gauche représente le numéro de la version où les requêtes ont matché depuis la version

n-1. Les données complètes et brutes sont également disponibles à l'Annexe A.

Tableau 5.3 – Résultats Broadleaf - Pourcentage de requêtes matchées par seuil choisi

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
1	21.74	21.74	21.74	21.74	8.7
2	100	100	100	81.82	27.27
3	45.45	45.45	45.45	45.45	36.36
4	34.62	34.62	30.77	19.23	7.69
8	100	100	100	100	97.67
9	30.61	30.61	30.61	30.61	20.41
10	10.64	10.64	10.64	10.64	0
32	100	100	100	100	86.76
39	7.14	7.14	7.14	7.14	5.71
42	6.94	6.94	6.94	6.94	2.78
43	5.06	5.06	5.06	5.06	3.8
46	100	100	100	100	96.43
53	7.53	7.53	7.53	7.53	1.08
55	15.05	15.05	15.05	6.45	0
57	18.95	18.95	18.95	18.95	3.16
58	32.65	32.65	32.65	31.63	18.37
59	7	7	7	5	4
63	25.51	25.51	25.51	25.51	24.49
65	17.39	17.39	17.39	17.39	10.43
66	22.61	22.61	22.61	22.61	15.65
75	4.51	4.51	4.51	4.51	3.76
76	4.51	4.51	4.51	3.76	3.76
77	23.31	23.31	23.31	23.31	22.56
80	5.65	5.65	5.65	5.65	0
86	9.68	9.68	9.68	6.45	0
89	24.79	24.79	24.79	24.79	18.8
92	2.7	2.7	2.7	2.7	0.9
93	5.41	5.41	5.41	4.5	1.8
94	11.82	11.82	11.82	11.82	10.91

Tableau 5.4 – Résultats OpenMRS - Pourcentage de requêtes matchées par seuil choisi

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
5	1.53	1.53	1.53	1.53	0.76
8	6.2	6.2	6.2	6.2	5.43
9	25	25	24.26	23.53	23.53
12	3.8	3.8	3.8	3.8	0
22	13.58	13.58	13.58	13.58	11.11
26	7.14	7.14	7.14	4.76	3.57
27	2.25	2.25	2.25	2.25	0
31	51.39	51.39	51.39	51.39	50
32	1.35	1.35	1.35	0	0
37	0.64	0.64	0.64	0	0
39	0.64	0.64	0.64	0.64	0
45	31.41	31.41	31.41	31.41	17.31
48	0.79	0.79	0.79	0.79	0
54	3.62	3.62	3.62	2.17	2.17
55	4.93	4.93	4.23	1.41	1.41
56	0.73	0.73	0.73	0.73	0
58	0.69	0.69	0	0	0
66	8.9	8.9	8.9	7.53	3.42
70	9.79	9.79	9.79	9.79	9.09
85	0.67	0.67	0.67	0	0
91	1.32	1.32	1.32	1.32	0
98	1.29	1.29	1.29	0.65	0.65
101	0.63	0.63	0.63	0	0
111	0.63	0.63	0.63	0.63	0
112	1.76	1.76	1.76	1.76	0
143	1.23	1.23	1.23	1.23	0.61
145	1.21	1.21	1.21	0.61	0.61
156	0.53	0.53	0.53	0.53	0

A partir de ces résultats, plusieurs enseignements intéressants sont à remarquer. Premièrement, il apparaît de manière évidente que plus le seuil minimum choisi est élevé, moins le nombre de matchings de requêtes détectés est important. Cela s'explique par le fait que le choix d'un tel seuil écarte prématurément lors des calculs des couples de requêtes qui auraient dû matcher au final. A contrario, le choix d'un seuil très bas permet d'éviter

ces erreurs, mais oblige à envisager énormément de combinaisons de couples superflus par ailleurs. Cela a donc pour conséquence une augmentation des temps de calcul.

Par conséquent, le choix d'un seuil minimum idéal consiste en un choix du compromis entre le temps de calcul et l'exhaustivité des combinaisons de requêtes devant matcher. Dès lors, les résultats des Tableaux font apparaître toutes versions confondues qu'un seuil fixé à 0.50 permet de répondre à ce compromis pour 218 versions sur 222, soit plus de 98% des versions. En effet, on constate sur les parties grisées des Tableaux le seuil jusqu'au quel le pourcentage de matchings détectés reste identique. Cela signifie donc que choisir le seuil qui couvre la partie grisée représente le choix idéal afin de ne manquer la détection d'aucun matching.

Cependant, il est intéressant de s'attarder sur les rares cas de versions où un seuil de 0.30 était nécessaire pour englober la totalité des matchings de requêtes, et ce pour en déterminer la cause. Le cas le plus important se réfère à la version 4 de BroadLeaf, où selon le Tableau 5.3, le pourcentage de requêtes matchées décroît de 34,62% à 30,77% du passage du seuil 0.30 au seuil 0.50, ce qui représente en fait la perte de détection de quatre requêtes. Ces quatre requêtes ont subi une évolution profonde due à un phénomène identique : deux classes dédiées aux DAO ont été refactorisées. En l'occurrence, la classe consacrée à la gestion des adresses des clients a été refactorisée pour séparer les informations qu'elle contenait au départ, à savoir l'état, le pays et l'adresse en elle-même. Un phénomène identique est également observé sur les informations de contact des clients, où l'email et le fax du client ont été séparés des numéros de téléphone. Ces refactorisations avaient donc finalement pour conséquence des requêtes accédant à plusieurs colonnes renommées très différemment (expliquant le seuil à 0.30 requis pour les détecter) pour les besoins des développeurs (cfr. Figure 5.1).

Finalement, cette situation particulière met en avant le fait qu'un seuil à 0.30 peut être utile dans certains cas spéciaux. En effet, lorsque les développeurs ont besoin d'effectuer de profondes modifications sur la logique du système qui impactent un grand nombre de colonnes, ce seuil minimum permet de s'assurer d'englober les cas de matchings les plus extrêmes. Cependant, on remarque que ceux-ci sont assez rares au final, au vu du nombre de requêtes

évoluées qui passent à la trappe au seuil de 0.50 . Une idée intéressante pour explorer complètement cette piste serait alors de faire varier le seuil choisi par version lorsqu'on détecte par exemple qu'une version paraît avoir été plus sensible à des modifications majeures.

Requête de la version 3	Requêtes liées de la version 4	Score H1	Score nom méthode	Score chemin
SELECT contact_id, customer_id, email, fax, primary_phone, secondary_phone FROM blc_contact_info WHERE customer_id = ?; (voir sur Github) **Chemin:** /BroadleafCommerce/src-profile /org/broadleafcommerce/profile /dao/ContactInfoDaoJpa.java **Méthode:** public List<ContactInfo> readContactInfoByUserId(Long customerId) **Technologie:** JPA	SELECT customer_phone_id, customer_id, phone_id, phone_name FROM blc_customer_phone WHERE customer_id = ? and customer_phone_id = ?; (voir sur Github) **Chemin:** /BroadleafCommerce/src-profile/org/broadleafcommerce /profile/dao/CustomerPhoneDaoJpa.java **Méthode:** public CustomerPhone readCustomerPhoneByIdAndCustomerId(Long customerPhoneId, Long customerId) **Technologie:** JPA	0.44680852	0.5068493	0.8518519

FIGURE 5.1 – Illustration de cas spécifique - Evolution avec seuil fixé à 0.30

5.3 Évaluation finale

La méthode présentée dans le Chapitre 4 fait finalement l'objet d'une évaluation afin d'en percevoir les points forts et les points faibles. Cette évaluation porte sur le cas d'étude du présent chapitre. Tout d'abord, la Section 5.3.1 présente les objectifs qui sont fixés pour l'évaluation, et la manière dont celle-ci se délimite. Ensuite, les résultats obtenus sur base de ce cas d'étude sont présentés dans la Section 5.3.2. Finalement, la Section 5.3.3 porte un regard critique sur l'évaluation établie.

5.3.1 Objectifs d'évaluation

Afin d'estimer la qualité de la méthode de détection d'évolution des requêtes, l'évaluation se concentre sur l'estimation de la précision des résultats d'évolution obtenus sur les deux systèmes présentés précédemment. Pour estimer ces résultats, plusieurs critères sont à considérer pour délimiter le champ d'évaluation.

Tout d'abord, un premier point important se réfère au fait qu'il n'existe pas un moyen initial basique afin de connaître avec certitude et à l'avance les évolutions de requêtes entre versions de système (ce qui explique par ailleurs la motivation d'en créer un dans le Chapitre 4). Dès lors, afin d'estimer si nos résultats sont corrects, il faut au préalable déterminer "à la main" quelles requêtes ont fait l'objet d'une évolution. Cette tâche étant assez fastidieuse et lourde à réaliser, cela réduit par conséquent l'évaluation à se concentrer sur un échantillon restreint des systèmes. Par conséquent, l'échantillon a été choisi selon ces critères afin de tenter d'être le plus représentatif possible :

A. L'échantillon possède 2 sous-échantillons à étudier : l'un se réfère à BroadLeaf, l'autre à OpenMRS.

B. Chaque sous-échantillon représente deux versions successives du système qui comportent un taux élevé (i.e. supérieur à 30%) de matchings de requêtes par rapport au nombre total de requêtes de ces versions.

C. Chaque sous-échantillon se réfère à une période relativement avancée dans le cycle de vie du système (i.e. au minimum quelques années de développement).

D. Les sous-échantillons à évaluer consistent en des matchings de deux requêtes, l'une issue d'une version, et l'autre de la version suivante. Dès lors et par exemple, un sous-échantillon de 100 requêtes correspond à 50 matchings à évaluer.

A partir de ces critères, le choix de l'échantillon final pour l'évaluation possède 396 requêtes, et donc 198 matchings. Ce total se décompose de la manière suivante : 101 matchings se réfèrent aux requêtes issues des versions 30 et 31 d'OpenMRS, tandis que les 97 autres matchings portent sur les requêtes des versions 57 et 58 de BroadLeaf. Le choix des seuils ayant été discuté dans la Section 5.2, la prise en compte d'un seuil fixé à 0.30 a été envisagé pour calculer les résultats faisant l'objet de l'évaluation.

Finalement, un dernier point à éclaircir intervient concernant l'évaluation. Notre méthode se portant garante dès le départ d'être une aide à la décision. Cela signifie que les matchings de requêtes calculés ne donnent pas une réponse binaire quant à l'évolution effective ou non. En effet, nous avons voulu que la méthode privilégie d'attribuer un pourcentage de similarité entre les requêtes qu'elle fait matcher, ce pourcentage aidant finalement l'utilisateur à déterminer lui-même si il considère qu'il y a évolution ou non. Or, dans l'optique d'évaluer la précision de détection d'évolution, nous sommes contraints de devoir fixer une valeur de pourcentage qui fait office de réponse binaire. Dès lors, il sera considéré durant l'évaluation qu'un pourcentage de matching supérieur à 80% indique formellement une évolution, tandis que toute valeur inférieure indique son contraire.

5.3.2 Résultats de l'évaluation

Sur base du cadre d'évaluation établi, les *vrai positifs* (i.e. VP) et les *faux positifs* (i.e. FP) ont été détectés à la main pour l'ensemble des matchings de l'échantillon choisi. A partir de cela, la précision de détection d'évolution est calculée selon la formule classique suivante :

$$Precision \equiv \frac{VP}{VP + FP} \tag{5.1}$$

Les résultats obtenus sont alors disponibles dans le Tableau 5.5. Sur les 198 matchings analysés, 7 d'entre eux s'avèrent être de faux positifs, ce qui procure une précision des résultats de 96%.

Tableau 5.5 – Résultats de l'évaluation - Version générale

Système	Vrai positif	Faux positif	Précision
OpenMRS	96	5	0,95
BroadLeaf	95	2	0,98
Total	**191**	**7**	**0,96**

Cependant, une remarque s'impose sur cet échantillon de résultats : les matchings de requêtes dont la valeur de similarité est de 100% sont inclues dedans. Autrement dit, cet échantillon comporte les cas d'évolution "triviale" où une requête est restée exactement la même entre les deux versions. Détecter ce type de cas est très important lorsque l'on souhaite parcourir l'entièreté du cycle de vie d'une requête tout au long du temps. Cependant, dans le cadre d'évaluation de précision des résultats, ces cas triviaux peuvent paraître trop simples à détecter par rapport aux autres, et donc venir finalement polluer l'évaluation de la précision de la méthode.

Par conséquent, le Tableau 5.6 présente une version plus stricte de l'évaluation, où le cas trivial des 100% est abandonné de l'échantillon pour se concentrer sur les autres évolutions. On remarque donc finalement sur cette évaluation plus stricte que la précision passe de 96% à 93% lorsqu'on évalue un échantillon de requêtes sans cas triviaux.

Tableau 5.6 – Résultats de l'évaluation - Version stricte

Système	Vrai positif	Faux positif	Précision
OpenMRS	68	5	0,93
BroadLeaf	31	2	0,94
Total	**99**	**7**	**0,93**

5.3.3 Regard critique sur l'évaluation

L'évaluation effectuée possède des qualités et des limites qu'il semble important de mentionner. Au niveau des limites, le fait de devoir vérifier "à la main" quelles requêtes ont effectivement évolué demande un travail assez fastidieux, ce qui oblige à réduire rapidement le champ d'investigation d'analyse des résultats. De plus, il est évidemment important de prendre du recul sur la valeur de précision obtenue pour cette raison, mais également sur le fait qu'il serait idéal de pouvoir disposer de plus que deux systèmes à comparer. Un point intéressant serait de poursuivre l'évaluation sur d'autres systèmes avec un historique déjà conséquent, et qui utiliseraient des technologies d'accès à la base de données ou des patterns différents de ceux utilisés par BroadLeaf et OpenMRS. Finalement, proposer d'utiliser la méthode dès le début de la phase de développement d'un système pourrait également apporter des bienfaits, permettant par exemple de disposer en fin de développement des données des développeurs sur les requêtes qui ont effectivement évolué, afin d'avoir un échantillon de requêtes plus large pour l'évaluation. Le bénéfice à en retirer du côté des développeurs est que la méthode outillée implémentée permet d'avertir des requêtes qui accèdent à des objets de la base de données périmés, ou encore de mettre en garde contre certaines requêtes copier-collées de manière abusive.

Un autre point négatif important concerne le *recall*, à savoir l'évaluation englobant les évolutions pertinentes qui n'ont pas été considérées par la méthode. En effet, l'évaluation de la *precision* ne permet pas d'avoir des informations à ce sujet. Cependant, le *recall* n'a pas pu être présenté dans l'évaluation pour plusieurs raisons. Premièrement, la méthode effectuant pour l'instant des matchings entre couple de deux requêtes, il est encore assez difficile pour cette dernière de pouvoir détecter certaines évolutions qui impliquent plus que deux requêtes isolées. Par conséquent, dans l'état actuel des choses, certains appronfondissements devraient encore être apportés sur cette problématique pour potentiellement permettre d'obtenir une valeur de *recall* honorable. Deuxièmement, il est difficile de récupérer à la main la globalité des évolutions pertinentes, a fortiori quand celles-ci se réfèrent à des évolutions qui impliquent plusieurs requêtes à la fois. Découvrir exhaustivement les évolutions est difficile dans un environnement de code qui n'est pas sous contrôle, ou autrement dit dans un système tiers qui n'est pas dédié à être

un système de laboratoire pour notre étude. Par conséquent, il semble peu rigoureux de calculer un *recall* sans pouvoir s'assurer avant que l'on connaît le nombre exact de faux négatifs.

Au niveau des points positifs, les versions choisies pour faire l'objet de l'évaluation l'ont été afin de trouver le sous-ensemble le plus représentatif des évolutions qui existaient sur l'ensemble de l'historique des systèmes. Dès lors, il paraît envisageable d'affirmer de manière assez prudente que les résultats obtenus seraient sensiblement identiques sur l'ensemble du système. Un autre point encourageant de l'évaluation est l'homogénéité des valeurs obtenues entre les deux systèmes différents : la précision obtenue est pratiquement identique, ce qui paraît plus rassurant qu'avoir des résultats complètement différents. En effet, des résultats différents indiqueraient probablement une hétérogénéité trop importante entre les sous-échantillons choisis, et donc des résultats trop spécifiques à des cas d'échantillons particuliers.

Finalement, les faux positifs détectés comportent souvent des cas spécifiques où un schéma identique d'erreur est fait par la méthode. Ainsi, dans le cadre de l'évaluation, on peut remarquer que la plupart des faux positifs consistaient en fait à ce que la méthode attribue un taux final de similarité important à cause par exemple du fait que les deux requêtes se trouvaient au même emplacement (i.e. le même fichier) alors qu'elles étaient très différentes au niveau du score de l'heuristique 1. Ce schéma d'erreur pourrait peut-être être corrigé en envisageant par exemple d'inclure la ligne de code où se situe la requête pour le calcul du score d'emplacement. De plus, les faux positifs comportaient systématiquement un taux de similarité proche des 80% là où les vrai positifs se situaient aux alentours des 90%, signe que la méthode restait plus prudente sur ces détections d'évolution erronées.

5.3.4 Cas de faux positifs

L'évaluation de la méthode outillée nous a permis d'identifier certaines situations particulières pouvant aboutir à des faux positifs. Il s'agit en fait de variations de la situation triviale dans laquelle une requête subit une modification trop conséquente que pour continuer à être détectée comme une version évoluée de cette requête : une requête semblable mais possédant déjà un matching d'évolution peut alors également être considérée comme une évolution de celle-ci. En effet, ce phénomène est autorisé par le fait que, de manière à prendre en compte le cas potentiel d'une fusion de requêtes, la méthode outiléée permet d'envisager l'évolution de plusieurs requêtes d'une version vers une même et seule requête de la version suivante.

La première situation observée apparaît donc lorsque qu'au moins deux requêtes d'une même version possèdent un haut niveau de similarité, et qu'une d'entre elles se voit modifiée dans l'autre version envisagée. Dans cette configuration, la requête modifiée n'est à tort pas détectée comme étant une version évoluée. En effet, d'autres requêtes sont alors considérées comme étant plus similaires, ce qui fausse les résultats. Les Figures 5.2 et 5.3 illustrent ce cas spécifique dans le système *BroadLeaf Commerce*. Dans la version *V = 8* (cf. Figure 5.2), la classe *"CustomerPhoneDaoImpl"* inclut les méthodes nommées *"readAllCustomerPhonesByCustomerId"* et *"readCustomerPhoneByIdAndCustomerId"*. Pour la version *V' = 9* (cf. Figure 5.3), nous pouvons observer que cette dernière méthode a disparu et a été remplacée par une nouvelle méthode avec un nom plus court mais similaire *"readCustomerPhoneById"*, impactant de la sorte la requête correspondante. Cependant, cette évolution n'est pas détectée par notre outil. De fait, dans la Figure 5.3 la requête exécutée à la *ligne 4* est maintenant moins similaire à la requête surlignée en *orange* dans la Figure 5.2 que la requête surlignée en *rouge*, ce qui aboutit ainsi à un faux positif.

```
1  public class CustomerPhoneDaoImpl implements CustomerPhoneDao {
2
3      public CustomerPhone readCustomerPhoneByIdAndCustomerId(Long
           customerPhoneId, Long customerId) {
4          Query query = em.createNamedQuery("
               BC_READ_CUSTOMER_PHONE_BY_ID_AND_CUSTOMER_ID");
5          query.setParameter("customerId", customerId);
6          query.setParameter("customerPhoneId", customerPhoneId);
7          List<CustomerPhone> customerPhones = query.getResultList();
8          return customerPhones.isEmpty() ? null : customerPhones.get(0);
9      }
10
11     // ...
12
13     public List<CustomerPhone> readAllCustomerPhonesByCustomerId(Long
           customerId) {
14         Query query = em.createNamedQuery("
               BC_READ_ALL_CUSTOMER_PHONES_BY_CUSTOMER_ID");
15         query.setParameter("customerId", customerId);
16         return query.getResultList();
17     }
18 }
```

FIGURE 5.2 – Illustration du premier cas de faux positif, version $V = 8$

```
1  public class CustomerPhoneDaoImpl implements CustomerPhoneDao {
2
3      public CustomerPhone readCustomerPhoneById(Long customerPhoneId) {
4          return (CustomerPhone) em.find(entityConfiguration.lookupEntityClass(
               CustomerPhone.class.getName()), customerPhoneId);
5      }
6
7      // ...
8
9      public List<CustomerPhone> readAllCustomerPhonesByCustomerId(Long
           customerId) {
10         Query query = em.createNamedQuery("
               BC_READ_ALL_CUSTOMER_PHONES_BY_CUSTOMER_ID");
11         query.setParameter("customerId", customerId);
12         return query.getResultList();
13     }
14 }
```

FIGURE 5.3 – Illustration du premier cas de faux positif, version $V' = 9$

La seconde situation observée dans le système *BroadLeaf Commerce* se déroule lorsqu'une requête initialement exécutée dans une méthode est ensuite exécutée dans une nouvelle méthode dont le rôle se limite exclusivement à un rôle d'intermédiaire. L'introduction de cette nouvelle méthode réduit sensiblement la probabilité d'évolution calculée par l'outil, ce qui fausse alors à nouveau le résultat si une autre requête semblable existe. Il s'agit du cas illustré dans les Figures 5.4 et 5.5. La requête exécutée dans la méthode *"readContentBySandbox"* est restée identique entre les deux versions. Ce n'est pas le cas de la requête construite dans la méthode *"readContentByIdsAndSandbox"*, qui est exécutée dans la version suivante (cf. Figure 5.5) dans un appel vers la méthode intermédiaire *"batchExecuteReadQuery"*. De par sa similarité, la requête exécutée à la ligne surlignée en *rouge* dans la version *V' = 31* est donc non seulement considérée comme une version identique par rapport à la version $V = 30$, mais aussi comme une évolution de la requête surlignée en orange dans la version $V = 30$, ce qui est erroné.

```
1  // V = 30
2  public class ContentDaoImpl implements ContentDao {
3
4      public List<Content> readContentByIdsAndSandbox(List<Integer> ids, String
           sandbox) {
5          Query query;
6
7          if (sandbox == null) {
8                  query = em.createNamedQuery("
                       BC_READ_CONTENT_BY_IDS_WHERE_SANDBOX_IS_NULL
                       ");
9                  query.setParameter("idList", ids);
10         } else {
11                 query = em.createNamedQuery("
                       BC_READ_CONTENT_BY_IDS_AND_SANDBOX");
12                 query.setParameter("idList", ids);
13             query.setParameter("sandbox", sandbox);
14         }
15
16         query.setHint(getQueryCacheableKey(), true);
17
18         return (List<Content>) query.getResultList() ;
19     }
20
21     // ...
22
23     public List<Content> readContentBySandbox(String sandbox) {
24         Query query = null;
25         if(sandbox!=null && sandbox.endsWith("*"))
26             query = em.createNamedQuery("
                       BC_READ_CONTENT_BY_LIKE_SANDBOX");
27         else
28             query = em.createNamedQuery("BC_READ_CONTENT_BY_SANDBOX");
29
30         query.setParameter("sandbox", sandbox);
31         query.setHint(getQueryCacheableKey(), true);
32
33         return (List<Content>) query.getResultList();
34     }
35 }
```

FIGURE 5.4 – Illustration du second cas de faux positif, version $V = 30$

```
1  // V' = 31
2  public class ContentDaoImpl extends BatchRetrieveDao implements ContentDao {
3
4      public List<Content> readContentByIdsAndSandbox(List<Integer> ids, String
           sandbox) {
5          Query query;
6
7          if (sandbox == null) {
8                  query = em.createNamedQuery("
                       BC_READ_CONTENT_BY_IDS_WHERE_SANDBOX_IS_NULL
                       ");
9          } else {
10                 query = em.createNamedQuery("
                       BC_READ_CONTENT_BY_IDS_AND_SANDBOX");
11             query.setParameter("sandbox", sandbox);
12         }
13
14         query.setHint(getQueryCacheableKey(), true);
15
16         return batchExecuteReadQuery(query, ids, "idList");
17     }
18
19     // ...
20
21     public List<Content> readContentBySandbox(String sandbox) {
22         Query query = null;
23         if(sandbox!=null && sandbox.endsWith("*"))
24             query = em.createNamedQuery("
                       BC_READ_CONTENT_BY_LIKE_SANDBOX");
25         else
26             query = em.createNamedQuery("BC_READ_CONTENT_BY_SANDBOX");
27
28         query.setParameter("sandbox", sandbox);
29         query.setHint(getQueryCacheableKey(), true);
30
31         return (List<Content>) query.getResultList();
32     }
33 }
```

FIGURE 5.5 – Illustration du second cas de faux positif, version *V' = 31*

Chapitre 6

Conclusion

6.1 Objectifs initiaux et aboutissements

Les systèmes d'informations évoluent de manière dynamique et constante, et doivent par ailleurs répondre constamment aux nouveaux défis d'évolution qui leur sont imposés. Dès lors, nous avons vu qu'il existe de nombreuses techniques pour supporter la maintenance, l'évolution logicielle voire également la migration de plate-formes, ces techniques étant situées à des niveaux de granularité différents. Cependant, au delà de ces techniques centrées particulièrement sur le présent et l'avenir du système, l'élément central qui nous a intéressé est de pouvoir analyser les riches ensembles de données du passé que constituent les dépôts logiciel. Le champ d'étude du *Mining Software Repositories* permet alors d'examiner rigoureusement et méthodiquement ces données qui reflètent l'historique du système. Comprendre le passé est souvent primordial pour détecter des erreurs potentielles et améliorer la qualité de fonctionnement future. Notre objectif initial a donc été de parvenir à élaborer de nouvelles manières d'analyser ces dépôts historiques pour mettre en lumière des informations qui y étaient cachées jusqu'à présent.

Afin de parvenir à cet objectif, analyser le cycle de vie des requêtes d'accès à la base de données peut permettre de retirer des informations nouvelles vis-à-vis des techniques de mining déjà existantes. Dès lors, la complétion de l'objectif s'est matérialisée par la création d'une nouvelle méthode d'analyse du cycle de vie des requêtes. Cette méthode repose sur la combinaison entre les travaux de Meurice et al. présentés dans le Chapitre 2 et une contribution personnelle exposée dans la suite du mémoire. Elle se décompose de la

manière suivante : le méta-modèle proposé par Meurice et al. (cf. Section 4.1.1) sert de fondation pour la récolte des informations relatives à l'extraction automatique des requêtes exécutées dans le code source du système et leur traduction dans une équivalence en langage SQL. A partir de cela, l'analyse du cycle de vie des requêtes SQL est établie via l'adaptation de critères emprunts au champ d'analyse d'origine d'artefacts du code sur base des travaux de Godfrey et al. [32]. Cette adaptation des critères était nécessaire afin de se centrer sur le contexte des requêtes, étant donné que les artefacts présentés par Godfrey et al. font initialement référence à des éléments tels que des fonctions comportant des appels entrants et sortants.

Finalement, la méthode présentée dans le Chapitre 4 a été outillée dans le but de l'évaluer sur base de deux cas concrets de systèmes open-source. La réalisation de cette évaluation a donc eu pour objectif de mesurer la précision d'analyse de la méthode, à savoir plus précisément la précision du nombre d'évolutions correctes de requêtes détectées. Pour y parvenir, la précision a été calculée sur un échantillon des versions des systèmes, dont les cas de faux positifs d'évolution de requêtes ont été détectés à la main au préalable. Les résultats obtenus font ressortir une précision d'analyse de requêtes matchées supérieure à 90% pour les deux systèmes. Les cas de faux positifs ont également fait l'objet d'une documentation spécifique, afin de pouvoir recenser quels types particuliers d'évolution nécessitent encore de devoir trouver une méthode de détection. Cette documentation est une piste importante à maintenir, étant donné qu'il existe de nombreuses manières différentes pour une requête d'évoluer, et que certaines d'entre elles cachent parfois des patterns plus compliqués à détecter qu'il n'y parait. Typiquement, lorsqu'une requête évolue sur base de plusieurs autres anciennes requêtes et non à partir d'une seule, cela fait apparaître des cas très spécifiques de modifications effectuées par les développeurs dans certaines situations.

6.2 Contraintes et limitations observées

Bien que les objectifs initiaux ont globalement pu être remplis pour un grand nombre de types d'évolution de requêtes, la méthode montre quelques faiblesses et limitations à remarquer. Tout d'abord, du point de vue purement strict de l'évaluation, avoir à disposition deux systèmes réellement utilisés à échelle raisonnable est certes un début très encourageant, mais reste un

nombre restreint qui limite à la prudence les conclusions pouvant être tirées des résultats obtenus. En effet, bien qu'un échantillon de requêtes le plus représentatif possible a été choisi, une telle méthode doit être utilisée sur un nombre plus élevé de systèmes pour pouvoir tirer des conclusions très solides. Malheureusement, le fait de devoir récolter à la main l'évolution effective des requêtes au préalable limite également fortement le champ d'action lors de l'évaluation. Une possibilité pour outrepasser cette contrainte pourrait être d'inclure une équipe développant un tout nouveau projet dans l'étude de cas, où les développeurs volontaires indiqueraient au fur et à mesure à la main les évolutions des requêtes. Cela permettrait alors d'obtenir un environnement idéal pour évaluer à grande échelle la méthode à la fin de la phase de développement, et ce de manière très précise. Le problème réside évidemment dans le fait qu'un tel environnement est à la fois difficile à mettre en place et long à se procurer.

Ensuite, une autre contrainte très importante à aborder concerne le passage à l'échelle d'une telle méthode outillée. En effet, bien qu'elle s'avère pour le moment être opérationnelle pour les cas abordés qui comportaient environ 300 000 lignes de code et une centaine de tables, le cas mentionné d'Oscar (qui s'élève à plus d'un million de lignes de code) a dû être abandonné pour des problèmes de volumétrie. Au niveau des temps de calcul, la complexité en temps reste problématique car la méthode telle qu'elle est proposée actuellement impose de devoir comparer absolument toutes les requêtes de chaque version entre elles, ce qui impose au minimum une complexité quadratique pour la comparaison de deux versions. Dès lors, cette limite impose de trouver des solutions ou des alternatives futures pour envisager de régler ces problèmes de volumétrie. Une piste à explorer pourrait par exemple consister à mettre en place du multithreading afin d'opérer du parallélisme au niveau des calculs d'heuristiques effectués.

Finalement, une autre contrainte notable concerne l'étendue des types de requêtes considérés par la méthode. Jusqu'à présent, celle-ci concentre son analyse sur des requêtes dont on fait l'hypothèse qu'elles ont été rédigées en langage SQL, ou qu'elles ont été traduites en SQL lorsqu'elles sont issues d'un autre langage tel que ceux que l'on retrouve dans le cas d'utilisation de technologies de mapping objet-relationnel. Cependant, pour rendre la méthode plus générale, il serait intéressant de l'étendre à diverses technologies d'accès

à la base de données sans passer par cette traduction nécessaire vers le langage SQL. Une généralisation des heuristiques développées jusqu'à présent devrait être alors envisagée pour y parvenir.

6.3 Perspectives futures

La méthode proposée ouvre diverses possibilités d'utilisation, notamment si les améliorations possibles décrites dans la section précédente sont prises en considération. Parmi les perspectives à envisager, il y a donc principalement deux cadres distincts qui se dégagent. D'une part, cette méthode vise à enrichir notre vision et notre analyse des données historiques présentes dans le dépôt logiciel du code source, et pourrait donc logiquement se combiner à l'analyse d'autres types de dépôts logiciel. Cela permettrait de croiser l'information pour en retirer des avantages. Ainsi et par exemple, il serait tout à fait envisageable de lier l'analyse du cycle de vie des requêtes à l'étude des rapports de bogues issus des dépôts de traçage de bogues. En effet, dans le domaine d'érosion du code, Eick et al. [18] proposent une série d'indices permettant de détecter l'érosion du code, parmi lesquels on retrouve entre autres l'historique des changements fréquents, la portée des changements, le potentiel d'erreur d'un module et l'effort nécessaire à l'implémentation d'un changement. Pour rappel, ces indices procuraient alors une aide pour prédire l'apparition d'une érosion de code, afin de pouvoir l'éviter. Par conséquent, utiliser la méthode d'analyse du cycle de vie des requêtes permettrait de détecter de nouveaux types de changements du code, et donc de renforcer les indices proposés par Eick et al. .

D'autre part, analyser le cycle de vie des requêtes peut permettre de proposer en amont des possibilités nouvelles dans le cadre de la maintenance de logiciels. En effet, combiner l'utilisation de la méthode avec l'analyse de l'évolution historique du schéma de la base de données pourrait permettre de faire de la détection d'erreurs, et servir de support à la maintenance des développeurs. Une manière d'effectuer cela serait par exemple de proposer en temps réel au développeur des localisations de portions de code contenant des requêtes qui nécessitent potentiellement d'être modifiées, sur base des changements subis par le schéma de la base de données. Cette perspective permettrait d'améliorer la qualité de gestion des modules d'accès à la base de données présents dans le programme. Ultimement, l'objectif serait alors

d'intégrer la méthode dans la gestion de la co-évolution entre le programme et sa base de données sur base de cela.

En conclusion, l'objectif recherché à travers la proposition d'une telle méthode n'est donc pas de donner une réponse isolée au développeur sur l'historique du système qui l'intéresse, mais d'ouvrir la voie à son utilisation pour entrecouper les résultats qu'elle procure avec d'autres techniques déjà existantes. Ces techniques peuvent être de l'ordre des recherches et contributions effectuées dans le cadre du *Mining Software Repositories* et de l'analyse du passé du système, ou plus globalement représenter des techniques de support à la maintenance qui, quant à elle, concerne bien plus souvent le présent ou le futur du système.

Bibliographie

[1] Lientz, B.P., & Swanson, E.B. (1980). *Software maintenance management : A study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley.

[2] Koskinen, J. (2003). *Software maintenance costs*.

[3] Seacord, R. C., Plakosh, D., & Lewis, G. A. (2003). *Modernizing legacy systems : software technologies, engineering processes, and business practices*. Addison-Wesley Professional.

[4] Jones, C. *Software Productivity Research*. (via Klint, P. (2011). *The Software Evolution Volcano*.)

[5] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., & Jazayeri, M. (2005, September). *Challenges in software evolution*. In Eighth International Workshop on Principles of Software Evolution (IWPSE'05) (pp. 13-22). IEEE.

[6] Lehman, M. M. (1979). *On understanding laws, evolution, and conservation in the large-program life cycle*. Journal of Systems and Software, 1, 213-221.

[7] Lehman, M. M. (1996, October). *Laws of software evolution revisited*. In European Workshop on Software Process Technology (pp. 108-124). Springer Berlin Heidelberg.

[8] Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997, November). *Metrics and laws of software evolution-the nineties view*. In Software Metrics Symposium, 1997. Proceedings., Fourth International (pp. 20-32). IEEE.

[9] Mattmann, C. A., Crichton, D. J., Hart, A. F., Goodale, C., Hughes, J. S., Kelly, S., ... & Medvidovic, N. (2011). *Architecting data-intensive software systems*. In Handbook of Data Intensive Computing (pp. 25-57). Springer New York.

[10] Gorton, I., Greenfield, P., Szalay, A., & Williams, R. (2008). *Data-intensive computing in the 21st century*. Computer, (4), 30-32.

[11] Lehman, M. M., Perry, D. E., & Ramil, J. F. (1998, November). *On evidence supporting the feast hypothesis and the laws of software evolution*. In Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International (pp. 84-88). IEEE.

[12] Godfrey, M. W., & Tu, Q. (2000). *Evolution in open source software : A case study*. In Software Maintenance, 2000. Proceedings. International Conference on (pp. 131-142). IEEE.

[13] D'Ambros, M., & Robbes, R. (2015). *Effective mining of software repositories*. Lecture. University of Chile. Santiago.

[14] D'Ambros, M., Gall, H., Lanza, M., & Pinzger, M. (2008). *Analysing software repositories to understand software evolution*. In Software evolution (pp. 37-67). Springer Berlin Heidelberg.

[15] Thomas, S.W., Hassan, A.E., & Blostein, D. (2014). *Mining unstructured software repositories*. In Evolving Software Systems, pp. 139–162.

[16] Moser, R., Pedrycz, W., & Succi, G. (2008, May). *A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction*. In 2008 ACM/IEEE 30th International Conference on Software Engineering (pp. 181-190). IEEE.

[17] Zimmermann, T., Premraj, R., & Zeller, A. (2007, May). *Predicting defects for eclipse*. In Predictor Models in Software Engineering, 2007. PROMISE'07 : ICSE Workshops 2007. International Workshop on (pp. 9-9). IEEE.

[18] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). *Does code decay ? assessing the evidence from change management data*. IEEE Transactions on Software Engineering, 27(1), 1-12.

[19] Zimmermann, T., Zeller, A., Weissgerber, P., & Diehl, S. (2005). *Mining version histories to guide software changes*. IEEE Transactions on Software Engineering, 31(6), 429-445.

[20] Agrawal, R., & Srikant, R. (1994, September). *Fast algorithms for mining association rules*. In Proc. 20th int. conf. very large data bases, VLDB (Vol. 1215, pp. 487-499).

[21] Göde, N., & Koschke, R. (2011, May). *Frequency and risks of changes to clones*. In Proceedings of the 33rd International Conference on Software Engineering (pp. 311-320). ACM.

[22] Zaidman, A., Van Rompaey, B., van Deursen, A., & Demeyer, S. (2011). *Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining*. Empirical Software Engineering, 16(3), 325-364.

[23] Hainaut, J. L. (2006). *The transformational approach to database engineering*. In Generative and transformational techniques in software engineering (pp. 95-143). Springer Berlin Heidelberg.

[24] Hainaut, J. L., Cleve, A., Henrard, J., & Hick, J. M. (2008). *Migration of legacy information systems*. In Software Evolution (pp. 105-138). Springer Berlin Heidelberg.

[25] Rahm, E., & Bernstein, P. A. (2006). *An online bibliography on schema evolution*. ACM Sigmod Record, 35(4), 30-31.

[26] Cleve, A., Gobert, M., Meurice, L., Maes, J., & Weber, J. (2015). *Understanding database schema evolution : A case study*. Science of Computer Programming, 97, 113-121.

[27] Hainaut, J. L. (2002). *Introduction to database reverse engineering*. LIBD Lecture Notes.

[28] Meurice, L., Nagy, C., & Cleve, A. (2016, June). *Static Analysis of Dynamic Database Usage in Java Systems*. In International Conference on Advanced Information Systems Engineering (pp. 491-506). Springer International Publishing.

[29] Meurice, L., Goeminne, M., Mens, T., Nagy, C., Decan, A., & Cleve, A. (2016). *Analyzing the Evolution of Database Usage in Data-Intensive Software Systems*.

[30] Meurice, L., Nagy, C., & Cleve, A. (2016). *Detecting and Preventing Program Inconsistencies Under Database Schema Evolution*. Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability & Security (QRS 2016). IEEE Computer society

[31] Linares-Vásquez, M., Li, B., Vendome, C., & Poshyvanyk, D. (2016, July). *Documenting database usages and schema constraints in database-centric applications*. In Proceedings of the 25th International Symposium on Software Testing and Analysis (pp. 270-281). ACM.

[32] Godfrey, M. W., & Zou, L. (2005). *Using origin analysis to detect merging and splitting of source code entities.* Software Engineering, IEEE Transactions on, 31(2), 166-181.

[33] Tornhill, A. (2015). *Your code as a crime scene.* Pragmatic Bookshelf, 35-47.

[34] Zimmermann, T., Nagappan, N., & Zeller, A. (2008). *Predicting bugs from history.* In Software Evolution (pp. 69-88). Springer Berlin Heidelberg.

Annexe A

Résultats du cas d'étude : données bruts

Tableau A.1 – Résultats Broadleaf - Pourcentage de requêtes matchées par seuil choisi

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
1	21.74	21.74	21.74	21.74	8.7
2	100	100	100	81.82	27.27
3	45.45	45.45	45.45	45.45	36.36
4	34.62	34.62	30.77	19.23	7.69
5	21.88	21.88	21.88	21.88	21.88
6	5.13	5.13	5.13	5.13	5.13
7	15	15	15	15	15
8	100	100	100	100	97.67
9	30.61	30.61	30.61	30.61	20.41
10	10.64	10.64	10.64	10.64	0
11	2.04	2.04	2.04	2.04	2.04
12	94.44	94.44	94.44	94.44	94.44
13	0	0	0	0	0
14	1.67	1.67	1.67	1.67	1.67
15	19.35	19.35	19.35	19.35	19.35
16	0	0	0	0	0
17	0	0	0	0	0
18	0	0	0	0	0
19	0	0	0	0	0
20	3.12	3.12	3.12	3.12	3.12
21	4.69	4.69	4.69	3.12	3.12
22	12.68	12.68	12.68	12.68	12.68
23	0	0	0	0	0
24	8.11	8.11	8.11	8.11	8.11
25	0	0	0	0	0
26	2.7	2.7	2.7	2.7	2.7
27	100	100	100	100	100
28	0	0	0	0	0
29	13.51	13.51	13.51	13.51	13.51
30	1.35	1.35	1.35	1.35	1.35
31	4.05	4.05	4.05	4.05	4.05
32	100	100	100	100	86.76
33	1.47	1.47	1.47	1.47	1.47
34	2.86	2.86	2.86	2.86	2.86
35	2.86	2.86	2.86	2.86	2.86

Tableau A.2 – Résultats Broadleaf - Pourcentage de requêtes matchées par seuil choisi (partie 2)

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
36	0	0	0	0	0
37	0	0	0	0	0
38	0	0	0	0	0
39	7.14	7.14	7.14	7.14	5.71
40	0	0	0	0	0
41	0	0	0	0	0
42	6.94	6.94	6.94	6.94	2.78
43	5.06	5.06	5.06	5.06	3.8
44	100	100	100	100	100
45	9.52	9.52	9.52	9.52	9.52
46	100	100	100	100	96.43
47	3.53	3.53	3.53	3.53	3.53
48	2.35	2.35	2.35	2.35	2.35
49	0	0	0	0	0
50	0	0	0	0	0
51	0	0	0	0	0
52	3.53	3.53	3.53	3.53	3.53
53	7.53	7.53	7.53	7.53	1.08
54	3.23	3.23	3.23	3.23	3.23
55	15.05	15.05	15.05	6.45	0
56	2.13	2.13	2.13	2.13	2.13
57	18.95	18.95	18.95	18.95	3.16
58	32.65	32.65	32.65	31.63	18.37
59	7	7	7	5	4
60	5.15	5.15	5.15	5.15	5.15
61	12.24	12.24	12.24	12.24	12.24
62	14.29	14.29	14.29	14.29	14.29
63	25.51	25.51	25.51	25.51	24.49
64	6.42	6.42	6.42	6.42	6.42
65	17.39	17.39	17.39	17.39	10.43
66	22.61	22.61	22.61	22.61	15.65
68	0	0	0	0	0
70	2.8	2.8	2.8	2.8	2.8

Tableau A.3 – Résultats Broadleaf - Pourcentage de requêtes matchées par seuil choisi (partie 3)

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
71	2.27	2.27	2.27	2.27	2.27
73	0	0	0	0	0
75	4.51	4.51	4.51	4.51	3.76
76	4.51	4.51	4.51	3.76	3.76
77	23.31	23.31	23.31	23.31	22.56
78	2.44	2.44	2.44	2.44	2.44
80	5.65	5.65	5.65	5.65	0
81	8.87	8.87	8.87	8.87	8.87
84	0.81	0.81	0.81	0.81	0.81
86	9.68	9.68	9.68	6.45	0
87	2.56	2.56	2.56	2.56	2.56
89	24.79	24.79	24.79	24.79	18.8
90	0	0	0	0	0
92	2.7	2.7	2.7	2.7	0.9
93	5.41	5.41	5.41	4.5	1.8
94	11.82	11.82	11.82	11.82	10.91

Tableau A.4 – Résultats OpenMRS - Pourcentage de requêtes matchées par seuil choisi

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	1.53	1.53	1.53	1.53	0.76
6	0	0	0	0	0
7	0	0	0	0	0
8	6.2	6.2	6.2	6.2	5.43
9	25	25	24.26	23.53	23.53
10	0	0	0	0	0
11	0	0	0	0	0
12	3.8	3.8	3.8	3.8	0
13	0	0	0	0	0
14	0	0	0	0	0
15	0	0	0	0	0
16	0	0	0	0	0
17	0	0	0	0	0
18	0	0	0	0	0
19	0	0	0	0	0
20	7.32	7.32	7.32	7.32	7.32
21	0	0	0	0	0
22	13.58	13.58	13.58	13.58	11.11
23	0	0	0	0	0
24	1.19	1.19	1.19	1.19	1.19
25	0	0	0	0	0
26	7.14	7.14	7.14	4.76	3.57
27	2.25	2.25	2.25	2.25	0
28	0	0	0	0	0
29	0	0	0	0	0
30	0	0	0	0	0
31	51.39	51.39	51.39	51.39	50
32	1.35	1.35	1.35	0	0
33	0	0	0	0	0
34	4.49	4.49	4.49	4.49	4.49
35	0	0	0	0	0

Tableau A.5 – Résultats OpenMRS - Pourcentage de requêtes matchées par seuil choisi (partie 2)

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
36	0.64	0.64	0.64	0.64	0.64
37	0.64	0.64	0.64	0	0
38	0	0	0	0	0
39	0.64	0.64	0.64	0.64	0
40	0.62	0.62	0.62	0.62	0.62
41	0	0	0	0	0
42	0	0	0	0	0
43	0	0	0	0	0
44	0	0	0	0	0
45	31.41	31.41	31.41	31.41	17.31
46	0	0	0	0	0
47	0	0	0	0	0
48	0.79	0.79	0.79	0.79	0
49	0	0	0	0	0
50	0	0	0	0	0
51	0.78	0.78	0.78	0.78	0.78
52	0	0	0	0	0
53	0	0	0	0	0
54	3.62	3.62	3.62	2.17	2.17
55	4.93	4.93	4.23	1.41	1.41
56	0.73	0.73	0.73	0.73	0
57	0	0	0	0	0
58	0.69	0.69	0	0	0
59	98.61	98.61	98.61	98.61	98.61
60	0	0	0	0	0
61	3.42	3.42	3.42	3.42	3.42
62	2.74	2.74	2.74	2.74	2.74
63	0	0	0	0	0
64	2.05	2.05	2.05	2.05	2.05
65	0	0	0	0	0
66	8.9	8.9	8.9	7.53	3.42
67	0	0	0	0	0
68	1.4	1.4	1.4	1.4	1.4
69	1.4	1.4	1.4	1.4	1.4
70	9.79	9.79	9.79	9.79	9.09

Tableau A.6 – Résultats OpenMRS - Pourcentage de requêtes matchées par seuil choisi (partie 3)

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
71	0.7	0.7	0.7	0.7	0.7
72	1.4	1.4	1.4	1.4	1.4
73	0.7	0.7	0.7	0.7	0.7
74	0.7	0.7	0.7	0.7	0.7
75	0.69	0.69	0.69	0.69	0.69
76	0	0	0	0	0
77	0	0	0	0	0
78	0	0	0	0	0
79	0.69	0.69	0.69	0.69	0.69
80	0	0	0	0	0
81	8.97	8.97	8.97	8.97	8.97
82	0.69	0.69	0.69	0.69	0.69
83	6.62	6.62	6.62	6.62	6.62
84	0.66	0.66	0.66	0.66	0.66
85	0.67	0.67	0.67	0	0
86	0.67	0.67	0.67	0.67	0.67
87	0	0	0	0	0
88	0	0	0	0	0
89	0	0	0	0	0
91	1.32	1.32	1.32	1.32	0
93	0.65	0.65	0.65	0.65	0.65
95	0.65	0.65	0.65	0.65	0.65
96	0	0	0	0	0
97	0	0	0	0	0
98	1.29	1.29	1.29	0.65	0.65
99	0	0	0	0	0
100	0	0	0	0	0

Tableau A.7 – Résultats OpenMRS - Pourcentage de requêtes matchées par seuil choisi (partie 4)

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
101	0.63	0.63	0.63	0	0
102	0	0	0	0	0
103	0	0	0	0	0
104	0	0	0	0	0
107	0.63	0.63	0.63	0.63	0.63
109	0	0	0	0	0
111	0.63	0.63	0.63	0.63	0
112	1.76	1.76	1.76	1.76	0
113	7.1	7.1	7.1	7.1	7.1
117	0	0	0	0	0
118	0	0	0	0	0
119	0	0	0	0	0
120	0	0	0	0	0
121	0	0	0	0	0
123	0	0	0	0	0
124	0	0	0	0	0
125	0	0	0	0	0
126	0	0	0	0	0
127	0	0	0	0	0
129	0	0	0	0	0
131	12.79	12.79	12.79	12.79	12.79
135	1.15	1.15	1.15	1.15	1.15

Tableau A.8 – Résultats OpenMRS - Pourcentage de requêtes matchées par seuil choisi (partie 5)

Version	Seuil 0.10	Seuil 0.30	Seuil 0.50	Seuil 0.70	Seuil 0.90
140	1.72	1.72	1.72	1.72	1.72
143	1.23	1.23	1.23	1.23	0.61
144	0	0	0	0	0
145	1.21	1.21	1.21	0.61	0.61
146	0	0	0	0	0
147	0	0	0	0	0
148	0.53	0.53	0.53	0.53	0.53
149	0	0	0	0	0
150	0	0	0	0	0
151	0	0	0	0	0
152	0	0	0	0	0
153	0	0	0	0	0
154	0.53	0.53	0.53	0.53	0.53
156	0.53	0.53	0.53	0.53	0
158	0	0	0	0	0
159	0	0	0	0	0
161	0	0	0	0	0
162	0	0	0	0	0
163	0	0	0	0	0